1

A

CANADIAN THESES ON MICROFICHE

I.S.B.N.

THESES CANADIENNES SUR MICROFICHE

National Library of Canada
Collections Development Branch

Canadian Theses on
Microfiche Service

Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Direction du développement des collections

Service des thèses canadiennes
sur microfiche

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us a poor photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de mauvaise qualité.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Division Division des thèses canadiennes

Ottawa, Canada
K1A 0N4

55592

PERMISSION TO MICROFILM — AUTORISATION DE MICROFILMER

• Please print or type — Écrire en lettres moulées ou dactylographier

Full Name of Author — Nom complet de l'auteur

SPIROS J BOUCOURIS

Date of Birth — Date de naissance

MAY 5, 1958

Country of Birth — Lieu de naissance

EGYPT

Permanent Address — Résidence fixe

341 MCLAREN ST Apt 903
OTTAWA ONTARIO CANADA

Title of Thesis — Titre de la thèse

A NOVEL ARCHITECTURE FOR A HIGH PERFORMANCE
DATA FLOW DIGITAL SIGNAL PROCESSOR

University — Université

CARLETON

Degree for which thesis was presented — Grade pour lequel cette thèse fut présentée

MASTER OF ENGINEERING

Year this degree conferred — Année d'obtention de ce grade

1982

Name of Supervisor — Nom du directeur de thèse

B. A BOWEN

Permission is hereby granted to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

L'autorisation est, par la présente, accordée à la BIBLIOTHÈQUE NATIONALE DU CANADA de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans l'autorisation écrite de l'auteur

Date

May 12, 1982

Signature

[Signature]

1

A NOVEL ARCHITECTURE FOR A HIGH PERFORMANCE
DATA FLOW DIGITAL SIGNAL PROCESSOR

by

Spiros J. Boucouris

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements
for the degree of
Master of Engineering.

Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University
Ottawa, Ontario

April 1982

21

The undersigned recommend to
the Faculty of Graduate Studies and Research
the acceptance of the thesis

"A Novel Architecture for a High Performance
Data Flow Digital Signal Processor"

by

Spiros J. Boutouris

in partial fulfillment of the requirements
for the degree of
Master of Engineering

Thesis Supervisor

Chairman,
Department of Systems
& Computer Engineering

Department of Systems and
Computer Engineering
Faculty of Engineering

May 1982

ABSTRACT

In this work an attempt is made to develop an architecture usable for the realization of high performance digital signal processors. The proposed architecture makes use of flow graphs, like the conventional data flow architectures, but has centralized control. The centralization of control becomes possible by taking into account some specific characteristics of digital signal processing algorithms. Additionally the proposed system is very flexible and easily expandable.

Care has been taken to avoid all possible bottlenecks either by completely removing their causes or by proposing ways to prevent them from occurring. Special care is given to the problem of control bottlenecks. A modular control unit is described which provides the necessary expandability without any possibility of saturation.

In order to validate the proposed architecture a simulation of the proposed system has been written in PASCAL and run on the PDP-11 computers available at the department. The simulation results, given in chapter 4, confirm the expected performance of the system. Then a discussion of the response time of the system is given. Finally a cost performance analysis is presented.

TABLE OF CONTENTS

ABSTRACT.....	1
PREFACE.....	3
GLOSSARY OF TERMS.....	6
1. DATA FLOW MACHINES	
1.1 Introduction.....	1-1
1.2 The Data Flow Concept.....	1-2
1.3 An ADA description.....	1-7
1.4 Data Flow Computers as DSPs.....	1-12
1.5 Architectural requirements for High Performance DSPs.....	1-14
1.5.1 Shared Memory Configurations.....	1-14
1.5.2 Distributed Memory Configurations.....	1-21
2. PROPOSED ARCHITECTURE FOR A HIGH PERFORMANCE DIGITAL SIGNAL PROCESSOR	
2.1 Introduction.....	2-1
2.2 General Properties of DSP Algorithms and Flow Graphs.....	2-2
2.3 General Overview of the Proposed System.....	2-5
2.4 Modules.....	2-10
2.5 Input / Output.....	2-17
2.6 Data Transfer Processors.....	2-19
2.7 Controller.....	2-20
3. AN ADA REPRESENTATION OF THE PROPOSED ARCHITECTURE.....	3-1
4. SYSTEM SIMULATION AND SIMULATION RESULTS	
4.1 Simulation.....	4-1
4.2 Simulation Results.....	4-6
5. SUMMARY AND CONCLUSIONS	
5.1 Comparison of the proposed architecture to previous architectures.....	5-1
5.2 System Cost versus Performance.....	5-2
5.3 System Response Time.....	5-5
5.4 Conclusions.....	5-11
REFERENCES.....	R-1
APPENDIX A	
Listing of Simulation Program.....	A-1

GLOSSARY OF TERMS

active element	An element than contains some intelligence and can act as a bus master (processor)
Hamming window	A windowing function $H(x) = 0.5 - 0.5 * \cos(x)$
microseconds	millionths of a second
module	An arithmetic processor with a set of private data memories
msec	thousandths of a second
multiport memory	A memory accessible from many ports with internal arbitration
multiprocessor	A system consisting of many processors
multiprocessing	The concurrent execution of many programs (not necessarily on many processors)
multiuser system	A system suporting many users concurrently
nsec	A billionth of a second
package	An ADA term. In its simplest form a package can be a pool of common data and type declarations. More generally, packages can be used to describe groups of related entities.
packet	A collection of instructions andt data combined according to some communications protocol
passive element	An element than acts always as a bus slave
rendezvous	An ADA term. It is the basic communication and synchronization mechanism for tasks. A task initiates an entry call and waits until the called task accepts the call.
supercomputer	A computer with throughput larger than today's mainframes

by several orders of magnitude
larger

task

An ADA term. Tasks are entities
that may operate in parallel.

template

The program unit of data flow
programs. It consists of an
OP-CODE, a number of operands
and one or more destination
addresses

throughput

The number of operations that
can be performed by a computer
in a unit of time

uniprocessor

A computer system consisting
of one processor

usec

microsecond

vs

versus

writable memory

RAM memory

ADA symbols used in Figures

ADA task

ADA package

Access to package

ADA rendezvous (only if immediately possible; caller does not wait)

ADA rendezvous (caller waits)

parameter passing

command passing

PREFACE

The requirements for digital signal processing have vastly increased during the past few years and are expected to increase even more in the future. In order to satisfy these requirements ultra fast computers, exploiting new architectures, will be needed with processing speed in the order of several billion operations per second.

Current architectural research is headed toward the development of complex parallel systems equipped with either adaptable or data flow architectures. The proliferation in both types of architecture was encouraged by advances in VLSI technology that significantly reduce the cost and enhance the reliability of computer components. Both adaptable and data flow architectures are capable of producing additional throughput from the available resources by organizing the computations more effectively than is done in existing systems.

Among the two, the data flow architecture has become more popular mainly because of the simplicity of the whole concept, the completely new point of view it introduces and the fact that it is technology transparent. However, the

data flow architecture seems to suffer from performance bottlenecks mainly due to the large number of data movements it requires.

In this work an attempt is made to develop an efficient data flow architecture specifically for digital signal processing applications. In chapter 1, a general overview of data flow machines is given. A discussion on the use of ADA constructs for the presentation of data flow machines is included along with a discussion of several alternative realizations of data flow machines as signal processors. Then, in chapter 2, a new architecture is presented. At first, the drawbacks of the previous architectures are pointed out, and then some properties of digital signal processing algorithms and flow graphs are discussed. Based on these, the new architecture is introduced and discussed in detail.

Chapter 3 contains a representation of the proposed architecture in ADA terms. This presentation is mainly focused on control flow and demonstrates once again the fact that ADA can be used for system description. In order to verify the feasibility of the proposed architecture a simulation program was written in PASCAL and run on the PDP-11 computers available at the department. The simulation results are presented in chapter 4. Chapter 5 contains a comparison of the proposed architecture to conventional architectures and a cost - performance analysis of the system.

ACKNOWLEDGEMENTS

I would like to thank the friends who have helped in this work and in particular I would like to thank, my thesis supervisor, Professor B.A.Bowen, who has been the main source of ideas and comments for this work.

April 1982

S.J.Boucouris

CHAPTER 1

DATA FLOW COMPUTERS

1.1 INTRODUCTION

The next generation of computers must provide efficient machines, with improved programmability, and high performance at acceptable cost. Additionally they must exploit the potential of LSI/VLSI technology. They must also be technology transparent, i.e., the architecture of future computers must be such that any new devices becoming available will be usable in the realization of a system without any need for major architectural changes. Data flow architectures offer a good solution to the problem of efficiently exploiting concurrency of computation on a large scale, are compatible with modern concepts of program structure, and seem to be fairly transparent to technology.

In this chapter the basic concepts of data flow ma-

chines will be presented, along with an evaluation of the architectural efficiency of data flow computers using ADA constructs, and their performance in signal processing applications.

1.2 THE DATA FLOW CONCEPT

The data flow concept is a fundamentally different way of looking at instruction execution in machine level programs. In other words it is an alternative to sequential instruction execution. In a conventional computer, an instruction is executed when the program counter reaches its location in memory. Clearly this limits the throughput of the system since any two completely independent instructions have to be executed in sequence. In a data flow computer, an instruction is executed when all its operands are available. There is no concept of control flow and there is no program counter. As a consequence many instructions of a data flow program may be available for execution and/or executed at the same time. Thus, highly concurrent computation is a natural result of the data flow concept.

Data flow computers are a form of language based architecture in which program graphs are the base language. A data flow program graph is made up of actors connected by

arcs. One kind of actor is the operator (figure 1.1) which also has input and output arcs carrying tokens with values. The arcs define paths over which values from one actor are conveyed by tokens to other actors. Tokens are placed on and removed from the arcs of a program graph according to firing rules. An actor is enabled when there are tokens on all input arcs and there is no token on any output arc. Any enabled actor may be fired. In the case of an operator, this means removing one token from each input arc, applying the specified function to the values associated with those tokens, and placing tokens labelled with the result value on the output arcs. Operators connected together form program graphs.

Another representation for data flow computers, much closer to the machine language used in prototype data flow computers, is the scheme in which a data flow program is represented as a collection of activity templates (figure 1.2), each corresponding to one or more actors of the data flow program graph. Each template contains an operation code, space for operands and destination address(es) for the result of the computation. The destination may be either another template or an I/O device.

A block diagram of a classical data flow machine is shown in figure 1.3. The fetch unit takes an instruction

address from the instruction queue, reads the corresponding activity template from the activity store, forms it into an operation packet and passes it to one of the processing elements in the operation unit. The processing element performs the operation specified by the template's operation code and generates one result packet for each destination field. The update unit receives result packets and enters the values they carry into operand fields of activity templates as specified by their destination fields. The update unit also tests if all operand and acknowledge packets required to activate the destination template have been received and, if so, enters the instruction address in the instruction queue.

Figure 1.1 An Operator Actor

Figure 1.2 An Activity Template

Figure 1.3 Basic Data Flow Machine

1.3 AN ADA DESCRIPTION

In this section the basic data flow machine described in section 1.2 will be represented in terms of ADA constructs such as tasks, packages and rendezvous. Some variations of the basic architecture will also be discussed [15].

The ADA representation illustrates the degree of concurrency possible in the data flow computer and provides some basis for discussion of system performance. Figure 1.4 illustrates a structure diagram representing, in ADA, the system in Figure 1.3. In ADA terms an active element can be directly represented as a task, while a passive element is a direct representation of an ADA package if it requires no control over access to it. However if controlled access to the element is required then the element is best represented as a server task which waits for other tasks to initiate rendezvous with it. For example the instruction queue of Figure 1.3 is represented as a server task since both the fetch and update tasks would require access to it, but for reasons of queue integrity, simultaneous access is not permitted. The activity store is represented as a package. This means that both fetch and update tasks are allowed simultaneous access to it. This is permissible because the fetch task only accesses templates that are ready to be operated on while the update task only accesses templates that

are not yet ready for processing. Since the update task decides which templates are ready it is guaranteed that the fetch and update tasks will never try to access the same information in the activity store simultaneously. Note that an additional task has been introduced into the system, namely the result packet queue. This task has been included to provide greater autonomy of the three processing tasks, thereby increasing the amount of concurrency and throughput in the system.

In order to illustrate that data flow mechanisms can be very conveniently expressed using ADA based design techniques we can consider the ADA representation of the actor shown in Figure 1.1. For clarity this same actor is also shown in Figure 1.5.a, while Figure 1.5.b shows its representation in ADA terms. Figure 1.6 shows a possible way for programming that actor to enforce the firing rules stated above. Here, nested rendezvous are used to enforce the rules. There is no intent to suggest that this is a practical implementation of such a low level operation; rather it is intended to show the logical equivalence between the two approaches. It is obvious from this example that the nature of data flow computing is very easily and directly expressed in ADA oriented structure diagrams and ADA programs.

Figure 1.4 Structure Diagram of a Data Flow Machine

Figure 1.5.a An Operator Actor

Figure 1.5.b Equivalent ADA Architecture of plus operator

```

TASK plus IS
    ENTRY put_x;
    ENTRY put_y;
END;

TASK BODY plus IS
    LOOP
        SELECT
            ACCEPT put_x(x)
            DO ACCEPT put_y(y)
                DO z := x + y;
                put_z(z);
            END;
        OR
            ACCEPT put_y(y)
            DO ACCEPT put_x(x)
                DO z := x + y;
                put_z(z);
            END;
        END SELECT;
    END LOOP;
END PLUS;

```

Figure 1.6 The PLUS task which Enforces the Data Flow Firing Rules

1.4 DATA FLOW COMPUTERS AS DSPs

A data flow signal processor must provide an instruction set consisting of basic signal processing operations. These common processing functions, which are designated as the basis operations of digital signal processing, are summarized in the following list [16] :

1. Difference Equations Calculations
2. Convolution
3. Correlation
4. Complex Frequency translation
5. Fast Fourier Transform
6. Inverse Fourier transform
7. Power Spectral Density or Magnitude Squared calculation
8. Matrix Arithmetic

9. Matrix Manipulation

10. Logarithms and Exponentiation

The signal processing applications served by these basis operations do not require recursive processing. Consequently recursion will not be considered in the data flow signal processor architectures.

An operation unit providing the basis operations requires instruction packets containing the following information :

- operation identification
- operation parameters (summation limits, coefficients)
- one or two (if required) input streams of data
- identification of destination instruction templates

In the next section a variety of configurations for a data flow signal processor are presented and described in ADA terms.

1.5 ARCHITECTURAL REQUIREMENTS FOR HIGH PERFORMANCE DIGITAL SIGNAL PROCESSORS

The architectural variations of the basic data flow computer which are presented in this section derive from two philosophies. One set of variations is based on the concept of a common memory accessed by all processing units, and the other on the concept of distributed memory owned by individual processing units.

1.5.1 SHARED MEMORY CONFIGURATIONS

The common view linking all the data flow architectures presented in this section is that the amount of data processed during any signal invocation of many of the basis functions is significant, to the point where the practice of passing data by value becomes unattractive.

For this reason, the signal processors considered here will have a common memory configuration. This view is valid as long as the possible memory contention bottlenecks have a significantly smaller impact on throughput than would the

communication overhead associated with the transfer of large blocks of data.

It is less clear whether operation parameters should be passed by value or by address. However, it seems reasonable to distinguish user data (data to be operated on) from instruction data (data that controls the operation). This, along with the fact that the amount of instruction data will be considerably smaller than the amount of user data, leads one to allocate user data in the common memory and instruction parameters to the activity store where it forms part of activity templates. This results in user data being passed by address in instruction and destination packets and in instruction data being passed by value.

However, when the system of figure 1.4 is applied to a signal processing application, it becomes clear that imbalances in processing time result. In particular, the operation unit becomes a serious bottleneck since the execution time of certain basis functions (e.g. DFT) is measured in milliseconds while the fetch and update activities can be performed in microseconds. A logical extension to the system of figure 1.4 would be to define multiple operation units working in parallel, as shown in figure 1.7. No change is made to the other tasks in the system. The number of additional operation units would be selected so that the

average time between the request of an instruction packet and the output of the result packet would approach the average processing time of the slower of the other two active tasks. Such a system would be very flexible in that it could handle all possible combinations of parallel function processing without loss of throughput.

This system has two main drawbacks. Firstly, the addition of operation unit accessing the same shared memory produces memory contention overhead so that the increase in throughput per additional operation unit is non-linear. Indeed, it is possible that memory bus saturation may occur before the desired throughput rate is achieved. The second drawback is cost. A multiport memory is required for the realization of this architecture. This also limits the expandability of the system since the simple addition of one more operation unit would require the addition of one more port to the memory. Besides, since each operation unit provides all basis functions, even at maximum throughput (i.e. all operation units busy) only 10% of the total functional capacity of the system is being used. Recall that the basis functions here are complex operations like FFTs, convolutions etc.

Another variation of the system, illustrated in Figure 1.8 deals with the issue of cost. Rather than defining a

single operation task for the entire set of basis functions, a separate task is defined for each function. This scheme provides increased performance over the basic system at lower cost than the system of Figure 1.7. (Implicit here is the assumption that the cost of splitting a single multiple-function task into many single-function tasks is significantly lower than the cost of providing many multiple function tasks). In addition to the definition of single function tasks, the system of Figure 1.8 exhibits another deviation from the previous ones: The separate Instruction_Queue task has been eliminated; instead the queue has been incorporated into the Fetch task. In fact, it was possible to do this in the previous systems with no loss of concurrency. Merging the two tasks becomes important in this system because of the existence of the single function tasks. Consider the situation where a single-instruction queue implemented as a ~~separate~~ task contains as its first two entries requests for the same function for which a single operation task exists. Once the first entry gets passed on to the appropriate task, the next entry in the instruction queue blocks all further processing until the request can be passed on. This prevents the processing of requests for other functions which may be available.

Figure 1.7 Common Memory Multiprocessor with many
Multiple Function Operation Units

Figure 1.8 Common Memory Multiprocessor with Single Function Operation Units

A solution to this problem is to define multiple instruction queues (one for each function) within the Fetch task itself. These queues are included within the Fetch task to avoid undesirable wait conditions that could arise if these queues were included in a separate task.

This implementation is advantageous in situations where the application(s) is well understood and the required number of tasks for each type of function can be predicted. In that case maximum utilization of the available resources without loss of throughput can be achieved. However, if the application is not well understood, it becomes much more difficult to assess the performance of this system than it would be for the previous one (Figure 1.7).

There is little doubt that the main bottleneck in the systems of Figures 1.7 and 1.8 is the contention for access to the shared data memory. Two remedies may be considered, although neither is very practicable in a system with many operation units. The first is to use multi-port memories, but there are severe limitations to the number of ports that can be placed on a single memory. The second, more radical approach is to build duplicate segmented memories configured so that each processor has exclusive write access to a segment of each memory. The write lines of the corresponding segments of each memory are tied together so that a write to

one memory segment will simultaneously write to the corresponding segment of all memories. The read lines are separated so that each processor has exclusive read access to its own copy of memory. This approach is clearly memory intensive; but it does relieve both read and write congestion. It would appear to be practical in an environment with only a limited number of processing units such that the costs of the duplicate memories would not become prohibitive.

1.5.2 DISTRIBUTED MEMORY CONFIGURATIONS

The distributed memory configurations considered in this section avoid the shared memory bottleneck problem by treating each operation unit as a separate processor with its own private memory. A communication path links the various elements of the system. The major drawback of this approach is that all data must now be passed by value throughout the system.

Figure 1.9 Distributed Memory Multiprocessor with Separate Communication Paths

Figure 1.9 shows one possible implementation of this approach. This system considers the Fetch, Update and Activity Store elements as part of a single processor (the instruction processor) connected via separate communication paths to many operation units. The same issues raised regarding common memory configurations, concerning the choice of multiple function or single function operation units, apply in this case. In effect, the system of Figure 1.9 is very close to those of Figures 1.7 and 1.8. The differences are that each operation unit has its own memory and that the communication paths are much busier therefore having a greater impact on system performance. Another disadvantage of this system is that the cost of separate links to each operation unit may become prohibitive.

Figure 1.10 is different from Figure 1.9 only in that the separate communication paths linking the individual operation units to the instruction processor are replaced by a single shared access path. This approach reduces system costs considerably but introduces overhead due to contention for the communication path.

Two considerations motivate a third possible distributed memory system. Firstly, the size of the activity store in a large, complex system may grow too large to be practical for a single memory activity store; it may be necessary

to distribute the activity store over many processor. Secondly, a desire for processor uniformity might dictate that each processor be a complete data flow processor in itself, i.e. that it contain all the elements of Figure 1.3. With this philosophy a distributed memory multiprocessor is achieved by interconnecting these units via a distributed network, as shown in Figure 1.11. It is not envisaged that the configuration of Figure 1.11 would ever be required on the basis of bottlenecks in the Fetch and Update elements. It is thought that in the signal processing environment, these elements would always be faster than the operation units. As in figure 1.10, the communication network may be considered in ADA terms as a middleman package.

Figure 1.10 Distributed Memory Multiprocessor with Shared Communication Path

Figure 1.11 Distributed Memory Multiprocessor with Multiple Instruction Stores

Summarizing the above we see that the advantage data flow machines have over other approaches to high performance computation is that the scheduling and synchronization of concurrent activities are built in at the hardware level, enabling each instruction execution to be treated as an independent concurrent action. This allows a high degree of parallelism which is limited when the synchronization and scheduling functions are realized in software or microcode.

Therefore, the data flow architecture seems a strong candidate for signal processing systems which are distributed and which have strong demands for modularity. An alternative architecture for achieving high performance will be discussed in the following chapters.

CHAPTER 2

PROPOSED ARCHITECTURE FOR A HIGH PERFORMANCE DIGITAL SIGNAL PROCESSOR

2.1 INTRODUCTION

The design of a high performance data flow digital signal processor includes design of control and communication mechanisms between very high speed VLSI components performing high level signal processing functions. These components are at the same logical level as tasks or processes and therefore ADA is very appropriate for representing the overall system architecture as a collection of cooperating components of this type.

In the previous chapter a variety of configurations for data flow signal processors have been presented and described in ADA terms. The basic uniprocessor data flow has

been considered inadequate for signal processing. Therefore configurations with multiple operation units were introduced. Two multiprocessor philosophies were investigated. The common memory multiprocessor system has the advantage of minimal data flow overhead but is susceptible to memory contention bottlenecks. The distributed memory multiprocessor suffers from no memory contention problems but is subject to considerably higher communication overhead. In both cases the major improvement in system throughput is expected to result from the addition of multiple operation units to the system.

In this chapter an alternative architecture will be discussed which seems to yield acceptable solutions to all the problems stated above. This architecture takes advantage of some properties of data flow graphs and of signal processing algorithms to yield a high performance system with relatively low cost and complexity.

2.2 GENERAL PROPERTIES OF DSP ALGORITHMS AND FLOW GRAPHS

This section will cover some properties of digital signal processing algorithms that will prove useful for the design of a digital signal processing system. By designing a

system with the specific applications in mind, a much higher performance for the specific application can be derived than in a system designed as a general purpose system and then adopted for digital signal processing.

First of all, most DSP algorithms are implemented to handle blocks of data rather than single samples. Thus in a multiprocessor digital signal processing system each processor will spend most of its time processing a block of data that does not need to be accessed by any other processor. This suggests the use of private memories, i.e. a system similar to the one shown in Figure 1.9. However such a system was considered to suffer from heavy bus load. This problem can be solved by using a number of dedicated data transfer buses. Thus the data need not go through the control unit, as is the case for the system in figure 1.9. A system designed with that in mind would have a much simpler interconnection and far less arbitration requirements than a common memory system while it would have many of its advantages.

Another important property of DSP algorithms is that they can easily be expressed in terms of a flow graph. Thus in the case of a processor having basis DSP functions each as one instruction, each node of the flow graph will correspond directly to one assembly language instruction. Since

most of these basis functions require the computation of sums of products it seems reasonable that a general purpose arithmetic processor could be designed to handle all basis functions with acceptable speed. In order to increase its speed the arithmetic processor may be equipped with some special purpose hardware for butterfly calculations, exponentiations, logarithms, etc. Furthermore, by choosing processors that can implement the whole set of basis functions, it is not necessary to move the data after the completion of an instruction. Instead, it is simpler to assign the execution of the next instruction in the flow graph to the same processor. This would not increase the overhead for control, but it would certainly decrease the loading of the bus(es).

Clearly this is applicable only if the graph is strictly sequential. At any point where there is a fork in the graph the movement of data cannot be avoided. Even in the case of a shared memory it would be necessary to generate a second copy of the data.

Figure 2.1.a shows a possible flow graph for an algorithm. The rectangles represent arithmetic operations (FFTs, Filters, etc) while the circles denote data movements. Figure 2.1.b shows the same flow graph separated into strictly sequential parts. All the templates connected with solid lines can be executed on the same processor without any data movements. The dotted lines denote that the operation will be triggered by the completion of the previous. The double circles denote data transfers that require some decision to be made by the controller. These symbols will be used throughout this work to denote the same things.

2.3 GENERAL OVERVIEW OF THE PROPOSED SYSTEM

In this section a general overview of the proposed architecture will be given. The proposed data flow machine will be described and presented from a logical point of view. Then, in the following sections, each of the elements that form the machine will be described in more detail. In

the following chapter a detailed ADA representation of the whole system will be given.

The main characteristic of the proposed system and its main difference from conventional data flow machines is the presence of a central controller that controls and synchronizes all the activities in the system. At a very high level the system can be thought of as a controller that directs the activities of a processing system (Figure 2.2). The controller accepts a program in the form of a flow graph, like the one shown in Figure 2.1, and then, based on that, it directs the activities of the processing system. Considering the fact that the execution times of most basis functions are in the order of milliseconds it is possible to have only one controller without bottlenecking the system.

Figure 2.1 - A typical flow graph

Figure 2.2 Block diagram of the proposed system

The processing system may consist of any number of arithmetic processors each of which capable of implementing all basis functions and each equipped with its own private memory. Each processor has access only to its own private memory. However all private memories are also accessible by one or more dedicated data transfer processors. These processors are also coordinated by the controller. The data transfer controller shown makes the number of data transfer processors transparent to the controller by acting as a coordinator between it and the data transfer processors. These data transfer processors handle all data transfers operating in parallel with the arithmetic processors that do the actual processing. This results in the 'crossbar' - like system shown in Figure 2.3.

The same data transfer processors also have access to the system I/O and handle all the required I/O. The actual I/O is handled locally by intelligent I/O interfaces that will input data and form buffers in their own private memory or receive buffers of data to be passed to an output device. In that respect an I/O interface looks very much like one of the modules.

Each of the elements of the system will be discussed in detail in the following sections of this chapter and some alternatives will be presented that could increase the flex-

ibility and throughput of the system without a parallel increase in price.

2.4 MODULES

"Module" is here used to mean an arithmetic processor with its private memory(ies). The specifications for the arithmetic processor will be discussed below. The private memory of the processor will also be accessible by the data transfer processors. However, it will not be accessible by any other arithmetic processor or the controller itself. Since the access to the memory by the arithmetic processor and/or the data transfer processors will only take place following commands from the controller, no complex synchronization mechanisms are necessary. However some arbitration to the access will be needed. This would increase the system cost and possibly limit the throughput due to memory contention. To avoid that, the private memory can be realized as a set of separate private memories instead of only one private memory (Figure 2.4).

Figure 2.3 Detailed block diagram of system

Figure 2.4 A Module

Figure 2.5 Realization of bus interconnections

Thus, the arithmetic processor may be processing data in any one memory at any time, while the data transfer processors will be transferring data to / from the other memories without causing any memory contention. Since all memory accesses will be directed by the controller, the nodes of the resulting cross-bar like system may be realized as simple transceivers (Figure 2.5). All memories discussed so far are strictly data memories. The program memory of each arithmetic processor is considered an integral part of the processor itself.

The most challenging part in the design of a module is the design of the arithmetic processor itself. A detail design of the processor is beyond the scope of this work. Here, only some necessary specifications for the processor will be given. An arithmetic processor intended to be used for signal processing must be capable of executing the basis signal processing functions in the minimum possible time. Clearly the fastest possible way of realizing the basis functions is by using dedicated hardware. This approach would result in a very expensive design and would lack flexibility. Besides, since it is desirable that all arithmetic processors will be capable of realizing all the basis functions it is obvious that only a small percentage of the hardware would be used at any time. This problem has also been mentioned in chapter 1 and lead to the use of separate

single function processors. However, it is possible to get a nearly optimum realization of high speed signal processing hardware by taking advantage of some characteristics of the signal processing algorithms. In the following discussion only integer arithmetic will be considered. This is by no means a limitation since the same approach can be used in the case of floating point arithmetic, including the use of special purpose VLSI hardware as it becomes available.

Most signal processing algorithms can be expressed as sums of products. Thus, a high speed multiplier should be the heart of a signal processor. However, this is not sufficient. Although most high speed multipliers are capable of doing accumulation of results, most algorithms require a larger number of additions than multiplications. Until recently the number of multiplications was considered to be the limiting factor, but after the realization of high speed multipliers it became obvious that the number of additions might cause a serious problem. Since the multiplier can make at most one addition per multiplication it is obvious that a separate adder is necessary. Instead of using a simple adder it would be better to use a dedicated ALU that would not only provide the necessary processing power for the required additions but would also serve for feeding the multiplier with data, would make any logical processing of data that might be necessary etc. That combination would be

capable of handling, very efficiently, most signal processing algorithms.

By including an ALU much greater flexibility and programmability of the arithmetic processor is achieved. The implementation of any special algorithm becomes a matter of writing the appropriate microcode. In other words, the arithmetic processors will not only be able to realize any basis function but it will also be possible to load them with some code to be executed for special applications. In order to extend the number of functions such a processor can implement, one need only add some more microprogram memory. Although only one of these functions will be executed at any time, the hardware utilization will always be high. Also, by including some writable control memory it becomes feasible to download custom algorithms for specific applications, while, by including some special hardware, a considerable reduction in the execution times of some basis functions can be achieved (FFT butterflies, logarithms etc).

This approach has already been used in some single chip digital signal processors. For example, the PD7720 signal processing chip introduced by NEC is built with an ALU and a multiplier operating in parallel. The chip contains some RAM and ROM memory for data and coefficient storage and some ROM for program storage. The arithmetic processor discussed

In this section is simply a larger version of the NEC chip including writable program storage and some special hardware for the computation of exponentials, logarithms, FFT butterflies etc. Therefore, it is reasonable to expect that such a processor will be available on a single chip within the next few years. Later, as more dense chips become available, it will be possible to realize a whole module on a single chip.

2.5 INPUT / OUTPUT

The system I/O will also be connected to the data transfer buses. Every I/O device will have an intelligent controller that will actually handle the device interface. The device controller will also be equipped with a set of memories like any arithmetic processor in the system (Figure 2.6). These memories will be used by the controller for temporary storage of data to be transferred to / from the device. These data will be transferred to / from the system by the data transfer processors.

Figure 2.6 An I/O module

By using this approach we can ensure that the system performance will not be limited by the I/O speed. Clearly the response time of any specific task will be heavily dependent on the required I/O, but the I/O delays will not have any effect on the system performance since the arithmetic processors will be running in parallel with the I/O handlers.

2.6 DATA TRANSFER PROCESSORS

Considering the structure of the I/O and the processing elements it becomes obvious that the data transfer processors will only be required to do memory to memory transfers. In other words, the data transfer processors need only be intelligent DMA controllers. This greatly simplifies the realization of these processors, since there are available today many intelligent DMA controllers with the required characteristics.

Furthermore, the interconnection of the memories makes it possible to get multiple copies of the same data during the same transfer, thereby reducing the amount of time required for the realization of a multiple fork in a flow graph. All that needs to be done for that is to enable more than one memory as destinations of the transfer. The only

other interconnection that has this advantage is the common memory system with duplicate segmented memory proposed in chapter one. However the cost and complexity of the system proposed here are considerably smaller. Also, the expandability of this system is much greater since any extra module can be connected directly to the existing data transfer buses and any extra data transfer processor can be connected in parallel to the existing ones with a few transceivers.

2.7 CONTROLLER

Since the operation of the whole system will be directed by a single controller, the design of the latter must be made with extreme care in order to avoid performance bottlenecks. As shown in Figure 2.2 the task of the controller is to take commands from the user, and, based on these commands, to direct the activities of the processing system. The first parameter of the system that must be specified before implementing the controller is the maximum possible control overhead. In order to clarify that we can refer again to Figure 2.1 showing a possible flow graph for an algorithm. The controller will accept the command to run this program and starting at the top node it will have to decide to which module to assign this work. The actual processing required by the controller at this point depends on how so-

phisticated the decision making algorithm will be. The simplest approach seems to be to assign the work to the module that presently has the minimum load (queue size). If only one type of processing is taking place in the system then the number of used memories in a module is a direct representation of the module's load. However, if many different types of processing are taking place, which is likely to be the case in a multiuser environment, then some more sophisticated criterion will be required. When that has been decided, the controller will assign the transfer of data from the I/O to the appropriate memory. When the data transfer processor acknowledges the completion of the transfer then the templates in the first subtask will be passed to the arithmetic processor for execution. This can be handled by a slave microprocessor and does not require the immediate attention of the controller. When the arithmetic processor acknowledges the completion of the assigned computation the controller must send the instruction for the generation of multiple copies of the data. To do that it must first decide to which module the data should be passed. The overhead for this operation is slightly more than that required for node 1 since now the decision must be made for two memories versus one for node 1. When that decision becomes available the steps described are repeated. As each subtask reaches its end, the controller is notified and commands the data transfer processors and the appropriate peri-

peripheral controllers to store the results if required. When this transfer is completed the controller will update its look-up tables to indicate that the memories used are available again. Additionally, the controller may have to pass some results by value, but the time required for this is negligible compared to the time required for the decision making.

Considering the relatively long time required for the execution of each template and the fact that the controller's decisions are mainly based on look-up tables, it becomes obvious that a single controller will suffice for a moderately large system. However, if the number of arithmetic processors in the system exceeds a limit then the controller will eventually bottleneck the system. In order to avoid that possibility and maintain the expandability of the system a two level control seems more appropriate.

In a two level controller a local controller is assigned for handling a certain number of arithmetic processors. The interface between the local controller and the modules is the same as the one described above. However, the local controller accepts its commands on tasks to be executed not directly from the user(s) but from a central controller. The central controller handles the interface to the user(s) and distributes the tasks more or less evenly

among the Local controllers. This approach removes any possibility for a bottleneck due to overloading of the controller since each local controller will have a limited number of modules to handle, while the central controller will only be concerned with distributing tasks. The only case where this approach would fail to meet the performance of the system with a single controller is at the execution of a program so large that the modules at the disposal of a local controller would be insufficient to handle it. This is not likely to happen since such a program should have an inherent parallelism larger than the number of arithmetic processors connected to a local controller. Figure 2.7 illustrates the two level control mechanism while Figure 2.8 shows the necessary changes that have to be made to the data transfer subsystem in order to incorporate the two-level control.

Figure 2.7 Two level control unit

Figure 2.8 Modified data transfer subsystem

CHAPTER 3

AN ADA REPRESENTATION OF THE PROPOSED ARCHITECTURE

In this chapter the ADA representation of the machine discussed in chapter 2 will be given. All the processors of the system are directly represented as ADA tasks while all memories in the system are directly represented as ADA packages. The latter representation may seem questionable since in a computer system no memory may be accessible by two or more processors simultaneously, while an ADA package may be accessed by any number of tasks. However, the representation holds true in this case since in the proposed system all memory accesses will be coordinated by the controller and therefore, there will never be any memory contention.

In order to achieve a complete and compact representation of the system in ADA terms, a number of middle-man tasks have been introduced. These tasks only accept calls from other tasks and in this particular case they only pass

data. The presence of these tasks helps achieve the maximum degree of concurrency possible for a given problem and also reduce the possibility of a deadlock situation. In practice, these tasks can be realized as a number of hardware FIFO queues directly accessible by the processors that call the middle-man task in the ADA description, or as a dedicated

microprocessor that will only communicate with the other processors in the system through FIFO queues. For reasons of exactness each of these FIFO queues should have been presented as a separate middle-man task. However, that would make the representation hard to understand and follow.

Therefore, wherever it is appropriate, a single middle-man task is used. This does not significantly change the performance of the system since the duties of the middle-man tasks are simple, and therefore their rendezvous have very short durations. Additionally the data transfer controller is represented as two separate tasks: One receiving the commands and distributing them to the data transfer processors, and one gathering the acknowledgements and directing them to the appropriate controller.

In this representation, the full machine with two levels of control will be given. The same representation holds for a simpler machine with only one level of control. In that case, one need only consider one local controller and

assume that it communicates directly with the user instead of the central controller. A number of figures help clarify the representation of the system in ADA terms.

The meaning of the parameters used in the representation is :

m	number of local controllers
n	number of arithmetic processors per controller
d	number of I/O devices
k	number of data transfer processors
r	number of memories per module

while Table 3.1 shows again a list of the symbols used in the ADA diagrams.

ADA task

ADA package

Access to package

ADA rendezvous (only if immediately possible; caller does not wait)

ADA rendezvous (caller waits)

parameter passing

command passing

Table 3.1 List of ADA symbols

```
MEMORY_SIZE : CONSTANT INTEGER := 4096;
```

```
TYPE USER_COMMAND IS RECORD
  -- programs and commands
END;
```

```
TYPE COMMAND_PACKET IS RECORD
  CNTRL : CONTROLLER_NUMBER;
  PROGRAM : TASK_INSTRUCTIONS;
END;
```

```
TYPE PROCESS IS RECORD
  TSKPTR : ^PROCESS;
END;
```

```
TYPE SUB_PROCESS IS RECORD
  SUBTSKPTR : ^TEMPLATE;
END;
```

```
TYPE MEMORY_TYPE IS ARRAY[1..MEMORY_SIZE] OF INTEGER;
```

```
PACKAGE MEMORYijq IS
  -- i = 1..m , j = 1..n ,
  -- q = 1..r
  MEMR : MEMORY_TYPE;
END;
```

Figure 3-1 Central Controller structure

```
TASK CENTRAL_CONTROLLER IS
  ENTRY USER_IN(COMMAND : USER_COMMAND);
```

```
TASK BODY CENTRAL_CONTROLLER IS
```

```
  BEGIN
    LOOP
      SELECT
        ACCEPT USER_IN(COMMAND : USER_COMMAND);
        PROCESS_COMMAND_AND_QUEUE_TASKS;
      OR
        WHEN NOT OUT_QUEUE_EMPTY =>
          DEQUEUE(PROCESS);
          CENTRAL_COMMAND_HANDLER.CENTRAL(PROCESS);
      OR
        CENTRAL_REPLY_HANDLER.CENTRAL(ANS);
        PROCESS_REPLY;
        IF MUST_REPORT
          THEN
            FORM_AND_END_REPORT;
          ELSE
            IF HELP_REQUESTED
              THEN BEGIN
                PROCESS_REQUEST_AND_FORM_COMMANDS;
                CENTRAL_COMMAND_HANDLER.CENTRAL(CMD);
              END;
            ELSE
              IF NOT TASK_QUEUE_EMPTY
                THEN
                  DEQUEUE(TSK);
            END SELECT;
          END LOOP;
        END;
```


TASK CENTRAL_COMMAND_HANDLER IS

```
ENTRY CENTRAL(CMD : COMMAND_PACKET);
ENTRY LOCAL1(TSK : PROCESS);
ENTRY LOCAL2(TSK : PROCESS);
ENTRY LOCAL3(TSK : PROCESS);
```

```
ENTRY LOCALn(TSK : PROCESS);
```

END;

TASK BODY CENTRAL_COMMAND_HANDLER IS

BEGIN

LOOP

SELECT

```
ACCEPT CENTRAL(CMD : COMMAND_PACKET);
PUT_IN_QUEUE(CMD);
```

OR

```
WHEN NOT EMPTY_QUEUE(1) =>
  ACCEPT LOCAL1(TSK : PROCESS);
  DEQUEUE(1,TSK);
```

OR

```
WHEN NOT EMPTY_QUEUE(2) =>
  ACCEPT LOCAL2(TSK : PROCESS);
  DEQUEUE(2,TSK);
```

OR

```
WHEN NOT EMPTY_QUEUE(3) =>
  ACCEPT LOCAL3(TSK : PROCESS);
  DEQUEUE(3,TSK);
```

OR

```
WHEN NOT EMPTY_QUEUE(m) =>
  ACCEPT LOCALm(TSK : PROCESS);
  DEQUEUE(m,TSK);
```

END SELECT;

END LOOP;

END;

TASK CENTRAL_REPLY_HANDLER IS

```

ENTRY CENTRAL(CMD : REPLY_PACKET);
ENTRY LOCAL1(REPL : REPLY);
ENTRY LOCAL2(REPL : REPLY);
ENTRY LOCAL3(REPL : REPLY);

```

```

ENTRY LOCALn(REPL : REPLY);

```

END;

TASK BODY CENTRAL_REPLY_HANDLER IS

```

BEGIN
  LOOP
    SELECT
      ACCEPT CENTRAL(REPL : REPLY_PACKET);
      DEQUEUE(REPL : REPLY_PACKET);
    OR
      ACCEPT LOCAL1(REPL : REPLY);
      FORM_PACKET_AND_QUEUE(1,REPL);
    OR
      ACCEPT LOCAL2(REPL : REPLY);
      FORM_PACKET_AND_QUEUE(2,REPL);
    OR
      ACCEPT LOCAL3(REPL : REPLY);
      FORM_PACKET_AND_QUEUE(3,REPL);
    OR
      ACCEPT LOCALm(REPL : REPLY);
      FORM_PACKET_AND_QUEUE(m,REPL);
    END SELECT;
  END LOOP;
END;

```

Figure 3.2 Local Controller structure

TASK LOCAL_CONTROLLERi IS . -- i = 1..m

TASK BODY LOCAL_CONTROLLERi IS

```

BEGIN
  LOOP
    SELECT
      CENTRAL_COMMAND_HANDLER.LOCALi(TSK);
      PROCESS_TASK_AND_QUEUE_SUBTASKS(TSK);
    OR
      WHEN NOT OUT_QUEUE_EMPTY =>
        DEQUEUE(CMD);
        LOCAL_COMMAND_HANDLERi.CNTR(CMD);
    OR
      WHEN NOT OUT_QUEUE_EMPTY =>
        DEQUEUE(REPL);
        CENTRAL_REPLY_HANDLER.LOCALi(REPL);
    OR
      LOCAL_REPLY_HANDLERi.OUT(ANS);
      ACTIVATE_SUBTASKS;
      IF MUST_REPORT
        THEN
          FORM_AND_QUEUE_REPLY;
        ELSE
          IF PROBLEM_TOO_LARGE
            THEN BEGIN
              FORM_HELP_REQUEST;
              CENTRAL_REPLY_HANDLER.LOCALi(REPL);
            END;
          ELSE
            IF NOT READY_QUEUE_EMPTY AND SPACE_AVAILABLE
              THEN BEGIN
                DEQUEUE(SUBTSK);
                IF IO_OPERATION
                  THEN BEGIN
                    SELECT_MODULE;
                    FORM_TEMPLATE_PACKET;
                    PUT_IN_OUT_QUEUE;
                  END;
                ELSE BEGIN
                  FORM_TEMPLATE_PACKET;
                  PUT_IN_OUT_QUEUE;
                END;
              END;
            END;
          END;
        END;
      END;
    END;
  END;

```

TASK LOCAL_COMMAND_HANDLERi IS

-- i = 1..m

```

ENTRY CNTR(CMD : TEMPLATE_PACKET);
ENTRY MODi1(TEMP : TEMPLATE);
ENTRY MODi2(TEMP : TEMPLATE);
ENTRY MODi3(TEMP : TEMPLATE);

```

```

--
ENTRY MODin(TEMP : TEMPLATE);
ENTRY DTPQ(TEMP : TEMPLATE);
ENTRY IOHND(TEMP : TEMPLATE);
END;

```

TASK BODY LOCAL_COMMAND_HANDLERi IS

```

BEGIN
  LOOP
    SELECT
      ACCEPT CNTR(CMD : TEMPLATE_PACKET);
      PUT_IN_APPROPRIATE_QUEUE;
    OR
      WHEN NOT EMPTY_QUEUE(1) =>
        ACCEPT MODi1(TEMP : TEMPLATE);
        DEQUEUE(1);
    OR
      WHEN NOT EMPTY_QUEUE(2) =>
        ACCEPT MODi2(TEMP : TEMPLATE);
        DEQUEUE(2);
    OR
      WHEN NOT EMPTY_QUEUE(3) =>
        ACCEPT MODi3(TEMP : TEMPLATE);
        DEQUEUE(3);
    --
    OR
      WHEN NOT EMPTY_QUEUE(n) =>
        ACCEPT MODin(TEMP : TEMPLATE);
        DEQUEUE(n);
    OR
      WHEN NOT EMPTY_DTPQUEUE =>
        ACCEPT DTPQ(TEMP : TEMPLATE);
        DEQUEUE(DTPQUEUE);
    OR
      WHEN NOT EMPTY_IOQUEUE =>
        ACCEPT IOHND(TEMP : TEMPLATE);
        DEQUEUE(IOQUEUE);
    END SELECT;
  END LOOP;
END;

```

```
TASK LOCAL_REPLY_HANDLERi IS      -- i = 1..m
```

```
ENTRY OUT(REP : LOCAL_REPLY_PACKET);
ENTRY MODi1(TEMP : LOCAL_REPLY);
ENTRY MODi2(TEMP : LOCAL_REPLY);
ENTRY MODi3(TEMP : LOCAL_REPLY);
```

```
ENTRY MODin(TEMP : LOCAL_REPLY);
ENTRY DTPQ(TEMP : LOCAL_REPLY);
ENTRY IORPL(TEMP : LOCAL_REPLY);
```

```
END;
```

```
TASK BODY LOCAL_REPLY_HANDLERi IS
```

```
BEGIN
```

```
LOOP
```

```
SELECT
```

```
ACCEPT OUT(REP : LOCAL_REPLY_PACKET);
DEQUEUE(REP);
```

```
OR
```

```
ACCEPT MODi1(TEMP : LOCAL_REPLY);
PACK_AND_QUEUE;
```

```
OR
```

```
ACCEPT MODi2(TEMP : LOCAL_REPLY);
PACK_AND_QUEUE;
```

```
OR
```

```
ACCEPT MODi3(TEMP : LOCAL_REPLY);
PACK_AND_QUEUE;
```

```
OR
```

```
ACCEPT MODin(TEMP : LOCAL_REPLY);
PACK_AND_QUEUE;
```

```
OR
```

```
ACCEPT DTPQ(TEMP : LOCAL_REPLY);
PACK_AND_QUEUE;
```

```
OR
```

```
ACCEPT IORPL(TEMP : LOCAL_REPLY);
PACK_AND_QUEUE;
```

```
END SELECT;
```

```
END LOOP;
```

```
END;
```

Figure 3.3 Input / Output Controller structure

TASK IO_COMMAND_HANDLER IS

ENTRY IOIN1(TEMP : TEMPLATE);
 ENTRY IOIN2(TEMP : TEMPLATE);
 ENTRY IOIN3(TEMP : TEMPLATE);

ENTRY IOINd(TEMP : TEMPLATE);

TASK BODY IO_COMMAND_HANDLER IS

```

BEGIN
  LOOP
    SELECT
      LOCAL_COMMAND_HANDLER1.IOHND(TMP);
      Q_TEMPLATE(TMP);
    OR
      LOCAL_COMMAND_HANDLER2.IOHND(TMP);
      Q_TEMPLATE(TMP);
    OR
      LOCAL_COMMAND_HANDLER3.IOHND(TMP);
      Q_TEMPLATE(TMP);

    OR
      LOCAL_COMMAND_HANDLERm.IOHND(TMP);
      Q_TEMPLATE(TMP);
    OR
      WHEN Q_NOT_EMPTY =>
        SELECT
          ACCEPT IOIN1(TMP : TEMPLATE);
          DEQUEUE(TMP);
        OR
          ACCEPT IOIN2(TMP : TEMPLATE);
          DEQUEUE(TMP);
        OR
          ACCEPT IOIN3(TMP : TEMPLATE);
          DEQUEUE(TMP);

        OR
          ACCEPT IOINd(TMP : TEMPLATE);
          DEQUEUE(TMP);
        END SELECT;
      END SELECT;
    END LOOP;
  END;

```


TASK IO_REPLY_HANDLER IS

ENTRY IOIN1(TEMP : TEMPLATE);
 ENTRY IOIN2(TEMP : TEMPLATE);
 ENTRY IOIN3(TEMP : TEMPLATE);

ENTRY IOINd(TEMP : TEMPLATE);

TASK BODY IO_REPLY_HANDLER IS

```

BEGIN
  LOOP
    SELECT
      WHEN NOT EMPTY Q1 =>
        LOCAL REPLY_HANDLER1.IORPL(TMP);
        Q_TEMPLATE(TMP);
      OR
        WHEN NOT EMPTY Q2 =>
          LOCAL REPLY_HANDLER2.IORPL(TMP);
          Q_TEMPLATE(TMP);
      OR
        WHEN NOT EMPTY Q3 =>
          LOCAL REPLY_HANDLER3.IORPL(TMP);
          Q_TEMPLATE(TMP);
      OR
        WHEN NOT EMPTY Qm =>
          LOCAL REPLY_HANDLERm.IORPL(TMP);
          Q_TEMPLATE(TMP);
      OR
        ACCEPT IOIN1(TMP : TEMPLATE);
        PUT_IN_Q(TMP);
      OR
        ACCEPT IOIN2(TMP : TEMPLATE);
        PUT_IN_Q(TMP);
      OR
        ACCEPT IOIN3(TMP : TEMPLATE);
        PUT_IN_Q(TMP);
      OR
        ACCEPT IOINd(TMP : TEMPLATE);
        PUT_IN_Q(TMP);
    END SELECT;
  END LOOP;
END;
```

Figure 3.4 Interconnection of Data Transfer Processors

TASK DTP_COMMAND_HANDLER IS

ENTRY DTPIN1(TEMP : TEMPLATE);
 ENTRY DTPIN2(TEMP : TEMPLATE);
 ENTRY DTPIN3(TEMP : TEMPLATE);

ENTRY DTPINK(TEMP : TEMPLATE);

TASK BODY DTP_COMMAND_HANDLER IS

BEGIN

LOOP

SELECT

LOCAL COMMAND_HANDLER1.DTPQ(TMP);
 Q_TEMPLATE(TMP);

OR

LOCAL COMMAND_HANDLER2.DTPQ(TMP);
 Q_TEMPLATE(TMP);

OR

LOCAL COMMAND_HANDLER3.DTPQ(TMP);
 Q_TEMPLATE(TMP);

OR

LOCAL COMMAND_HANDLERm.DTPQ(TMP);
 Q_TEMPLATE(TMP);

OR

WHEN Q_NOT_EMPTY =>

SELECT

ACCEPT DTPIN1(TMP : TEMPLATE);
 DEQUEUE(TMP);

OR

ACCEPT DTPIN2(TMP : TEMPLATE);
 DEQUEUE(TMP);

OR

ACCEPT DTPIN3(TMP : TEMPLATE);
 DEQUEUE(TMP);

OR

ACCEPT DTPINK(TMP : TEMPLATE);
 DEQUEUE(TMP);

END SELECT;

END SELECT;

END LOOP;

END;

TASK DTP_REPLY_HANDLER IS

ENTRY DTPIN1(TEMP : TEMPLATE);
 ENTRY DTPIN2(TEMP : TEMPLATE);
 ENTRY DTPIN3(TEMP : TEMPLATE);

ENTRY DTPINK(TEMP : TEMPLATE);

TASK BODY DTP_REPLY_HANDLER IS

BEGIN
 LOOP
 SELECT
 WHEN NOT EMPTY Q1 =>
 LOCAL_REPLY_HANDLER1.DTPQ(TMP);
 Q_TEMPLATE(TMP);
 OR
 WHEN NOT EMPTY Q2 =>
 LOCAL_REPLY_HANDLER2.DTPQ(TMP);
 Q_TEMPLATE(TMP);
 OR
 WHEN NOT EMPTY Q3 =>
 LOCAL_REPLY_HANDLER3.DTPQ(TMP);
 Q_TEMPLATE(TMP);
 OR
 WHEN NOT EMPTY Qm =>
 LOCAL_REPLY_HANDLERm.DTPQ(TMP);
 Q_TEMPLATE(TMP);
 OR
 ACCEPT DTPIN1(TMP : TEMPLATE);
 PUT_IN_Q(TMP);
 OR
 ACCEPT DTPIN2(TMP : TEMPLATE);
 PUT_IN_Q(TMP);
 OR
 ACCEPT DTPIN3(TMP : TEMPLATE);
 PUT_IN_Q(TMP);
 OR
 ACCEPT DTPINK(TMP : TEMPLATE);
 PUT_IN_Q(TMP);
 END SELECT;
 END LOOP;
 END;

Figure 3.5 Interconnection of Arithmetic Processors

```

TASK ARITHMETIC_PROCESSORij IS          -- i = 1..m , j = 1..n
TASK BODY ARITHMETIC_PROCESSORij IS    -- i = 1..m , j = 1..n

```

```

BEGIN
  LOOP
    LOCAL COMMAND_HANDLERi.MODij(TMP);
    EXECUTE_TEMPLATES;
    FORM_REPLY;
    LOCAL REPLY_HANDLERi.MODij(REP);
  END LOOP;
END;

```

```

TASK DATA_TRANSFER_PROCESSORp IS      -- p = 1..k

```

```

TASK BODY DATA_TRANSFER_PROCESSORp IS  -- p = 1..k

```

```

BEGIN
  LOOP
    DTP_COMMAND_HANDLER.DTPINp(TMP);
    EXECUTE_TRANSFER;
    DTP_REPLY_HANDLER.DTPINp;
  END LOOP;
END;

```

```

TASK IO_CONTROLLERc IS                -- c = 1..d

```

```

TASK BODY IO_CONTROLLERc IS          -- c = 1..d

```

```

BEGIN
  LOOP
    IO_COMMAND_HANDLER.IOINC;
    PERFORM_IO;
    FORM_REPLY(REP);
    IO_REPLY_HANDLER.IOINC(REP);
  END LOOP;
END;

```

CHAPTER 4

SYSTEM SIMULATION AND SIMULATION RESULTS

4.1 SIMULATION

In order to check the feasibility of the proposed system as well as test its performance under various types of loading conditions a simulation program was written in PASCAL and run on the PDP-11/55 and PDP-11/60 computers available at the department. A listing of the program is given in Appendix A.

The program is a logical simulation of the system described in chapter 2. The number of processors in the system can be varied along with the maximum number of memories per module and the number of data transfer processors. Two types of jobs are considered. One is called a "real time" job while the other is called a "nonreal" time job. "Real time" job here means a job initiated at fixed time intervals. In other words the system receives requests for the execution of this job at fixed time intervals. This time

interval can be thought as the mean of the random length time intervals that would be the case in a real system. The non-real time jobs are considered to be always present as a continuous load to the system. These jobs run as soon as the required resources become available and then, after their completion, they are placed in the input queue again. The number of nonreal time jobs that are allowed to be in the system at any time can be varied. By using that approach it is possible to observe the operation of the system under different load conditions. These results will be presented and discussed in the following section.

The graphs of the programs used to derive the results presented in this chapter are shown in Figures 4.1 and 4.2. The graph shown in Figure 4.1 corresponds to the nonreal time job used. The number of nonreal time jobs allowed in the system was set to twice the number of arithmetic processors in the system. That minimized the idle time of the arithmetic processors since by the time the processor had completed one program the data for the next one would have already been loaded into one of its memories. The graph of Figure 4.2 represents the real time job. In this case the graph has to be divided into strictly sequential parts. Upon completion of part 1 the data are copied and parts 2 and 3 are triggered. The time interval between the initiation of two successive jobs was chosen so that the total

load would not exceed the total capacity of the arithmetic processors and/or the total capacity of the data transfer processors.

Some test runs were made where the requested processing exceeded the capacity of the system. When the data transfer processors were the bottleneck then an input queue was formed, the size of which was continuously increasing. On the other hand, when the load exceeded the capacity of the arithmetic processors, then at first the data were accumulated in the available memories. When all the memories were filled then the same input queue was formed as in the previous case.

Considering the reduced cost of memory and the continuing reduction estimated for the future it becomes obvious that the memory size would never be allowed to bottleneck the system. Therefore the number of available memories per module was set to a number large enough to ensure that the memory would never become a bottleneck. The program counts how many memories were used at each module at any time and after the end of the simulation it reports the largest number of memories used at any time. In other words this program can also be used for the design of a system based on the proposed architecture that would be capable of handling some specific types of processing.

Figure 4.1 Flow Graph of non-real time job

Figure 4.2 Flow Graph of real time job

4.2 SIMULATION RESULTS

The results of the simulation runs for different load conditions are shown in Figures 4.3 through 4.10. Figures 4.3 through 4.6 correspond to a load of nonreal time jobs while Figures 4.7 through 4.10 to a load of real time jobs. From these Figures it becomes obvious that the way the system responds does not depend on the nature of the load although the actual parametric limits are heavily dependent on the load. For programs that require a large number of transfers but relatively few computations the knee of the throughput curve occurs earlier than in the case that the programs require a relatively smaller number of transfers. That is because, in the first case, the data transfer processors are soon loaded to capacity and therefore they become the bottleneck of the system. Because of that, the addition of more processing elements will not increase the system throughput, since the extra modules that will be added will practically be idle due to the lack of data to be processed. Furthermore, the addition of these elements will increase the load of the system controller, and therefore, it is possible that the addition of more processing elements will actually reduce the system throughput.

Figures 4.3 and 4.7 show a plot of the system throughput as a function of the number of processors with the

number of data transfer processors as a parameter. The throughput is calculated in multiplications per second. In this simulation some reasonable execution times have been assumed for each of the basis functions (Table 4.1). Also, the approximate number of multiplications required for each basis function has been calculated (Table 4.2). The times assumed correspond to existing implementations of the basis functions with specially configured hardware.

It is important to note the sharp transition of the function from a sloped line to a horizontal line. This sharp transition indicates that it is possible to get the system running to almost theoretical throughput as long as the data transfer processors do not bottleneck the system.

Basis Function	Execution Time (msec)
Hamming Window	0.15
Frequency Shift	8.00
FFT	6.00
Magnitude Squared	0.50
Matched Filter	10.00
Convolution	20.00

Table 4.1 Execution times of basis functions

Basis Function	Number of Multiplications (K)
Hamming Window	1.0
Frequency Shift	25.0
FFT	20.0
Magnitude Squared	2.0
Matched Filter	28.0
Convolution	65.0

Table 4.2 Number of multiplications required for each basis function

Figures 4.4 and 4.8 show the utilization of the processing elements as a function of their number with the number of data transfer processors as a parameter. It can be seen that the system runs at maximum utilization regardless of the number of processors as long as the data transfer processors do not become a bottleneck. When that happens the utilization of the processors decreases in a hyperbolic fashion. This is to be expected since the amount of data to be processed, and therefore the actual processing taking place, is limited to the capacity of the data transfer processors. Therefore, the utilization of the system follows the expression

$$U(N) = \text{const} / N$$

where N is the number of processing elements and const is a constant depending on the number of data transfer processors and the type of processing that takes place.

Figures 4.5 and 4.9 show the utilization of the data transfer processors as a function of the number of arithmetic processors in the system again with the number of data transfer processors as a parameter. It can be seen that the utilization of the data transfer processors increases linearly with the number of processors in the system until it reaches a maximum, and after that it remains un-

changed. A small overshoot appears in the utilization curve, shortly after the maximum utilization has been reached, but it cannot be used as an optimum point since at by this time the utilization of the arithmetic processors, as shown in figure 4.8 has already decreased considerable. The maximum utilization of the data transfer processors is about 95%. This maximum is the result of the times assumed for the system and will change if a different timing is actually used. The assumptions made were that the data transfer processors operate from a 20MHz clock and each transfer requires four clock cycles. The time required for the initial setup of the data transfer processor, after it has received a command, is assumed to be 10 microseconds, while all transfers are assumed to be 1024 word transfers.

Combining the above :

$$4 / 20\text{MHz} \Rightarrow 200 \text{ nsec} / \text{word transfer}$$

$$0.2 * 1024 \Rightarrow 204.8 \text{ usec} / \text{block transfer}$$

$$204.8 + 10 \Rightarrow 214.8 \text{ usec} / \text{block transfer}$$

(including overhead)

$$204.8 / 214.8 \Rightarrow 95.35 \% \text{ maximum utilization}$$

The maximum utilization of the data transfer processors as

computed by the simulation program is 95.22%. By studying Figures 4.3 and 4.7 along with Figures 4.5 and 4.9 respectively we see that the system reaches saturation when the utilization of the data transfer processors has reached its maximum. After this point the system throughput remains approximately constant, limited by the data transfer processors, while the utilization of the arithmetic processors decreases as shown in Figures 4.4 and 4.8.

Finally, Figures 4.6 and 4.10 show the control requirements of the system. As expected the shape of these curves is identical to that of the throughput curves. Once again, the exact parametric limits depend on the type of the load under which the system is operating. The control requirements for real time jobs are slightly higher than those for nonreal time jobs. This could be predicted by simply calculating the execution times of these jobs. The time required for the recognition of an instruction by the arithmetic processor, called setup time, was estimated to be 10 microseconds. For system operation with full utilization of the arithmetic processors, we have for the nonreal time jobs

Execution time on single processor =

$$T = t(\text{Hamming Window}) +$$

$$\begin{aligned}
 & t(\text{Frequency Shift}) + \\
 & t(\text{FFT}) + \\
 & 3 * \text{setup time} \\
 & = 0.15 + 8.0 + 6.0 + 3 * 0.01 \\
 & = 14.18 \text{ msec.}
 \end{aligned}$$

while for the real time jobs we have

$$\begin{aligned}
 T &= t(\text{Hamming Window}) + \\
 & t(\text{Frequency Shift}) + \\
 & t(\text{Matched Filter}) + \\
 & t(\text{FFT}) + \\
 & t(\text{Magnitude Squared}) + \\
 & 5 * \text{setup time} \\
 & = 0.15 + 8.0 + 10.0 + 6.0 + 0.5 + 5 * 0.01 \\
 & = 24.70 \text{ msec}
 \end{aligned}$$

The number of decisions the controller (the whole control system) has to make every second for a given type of job is given by the formula :

$$D = N_d * N / T$$

22
OF / DE

A

where

D is the number of decisions

N_d is the number of decisions per job

N is the number of arithmetic processors in the system

T is the execution time of the job on a single processor

For the nonreal time jobs we have

$$D = 1 * N / 0.01418$$

$$= 70.52 * N$$

while for the real time jobs

$$D = 2 * N / 0.0247$$

$$= 80.97 * N$$

The calculated slope for both the nonreal time and real time jobs are very close to the slopes in Figures 4.6 and 4.10 respectively. It is important to note that these number only give a measure of the number of decision that need to be made and not the processing required to actually make them. In a real system, this time will depend on the type of processor used as controller and the control algorithm employed. For the simulation, a very simple algorithm has been used. Any new job was assigned to the processor that currently had the largest number of free memories. However, in a real system a more sophisticated control algorithm will be needed to avoid some deadlocks that may occur if the system is overloaded. Once a controller has been chosen it will be straightforward to decide how many arithmetic processors can be handled by each controller without any performance bottlenecks.

Figure 4.3 System throughput vs number of processors
for the program in figure 4.1.

The numbers on the curves indicate the number of data transfer processors. The \$ signs indicate points where more than one value coincide

Figure 4.4 Arithmetic processors utilization
for the program in figure 4.1
vs number of processors

The numbers on the curves indicate the number of data transfer processors. The S signs indicate points where more than one value coincide.

Figure 4.5 Data Transfer Processors utilization
for the program in figure 4.1
vs number of processors

The numbers on the curves indicate the number of data transfer processors. The \$ signs indicate points where more than one value coincide.

Figure 4.6 Control requirements (overhead)
for the program in figure 4.1
vs number of processors

The numbers on the curves indicate the number of data transfer processors. The \$ signs indicate points where more than one value coincide.

Figure 4.7 System throughput vs number of processors
for the program in figure 4.2

The numbers on the curves indicate the number of data transfer processors. The \$ signs indicate points where more than one value coincide.

Figure 4.8 Arithmetic processors utilization
for the program in figure 4.2
vs number of processors

The numbers on the curves indicate the number of data transfer processors. The \$ signs indicate points where more than one value coincide.

Figure 4.9 Data Transfer Processors utilization
for the program in figure 4.2
vs number of processors

The numbers on the curves indicate the number of data transfer processors. The \$ signs indicate points where more than one value coincide.

Figure 4.10 Control requirements (overhead)
for the program in figure 4.2
vs number of processors

The numbers on the curves indicate the number of data transfer processors. The \$ sign indicates points where more than one value coincide.

CHAPTER 5

SUMMARY AND CONCLUSIONS.

5.1. ~~COMPARISON~~ OF THE PROPOSED ARCHITECTURE TO PREVIOUS ARCHITECTURES.

In this section, a comparison of the proposed architecture to previous architectures, both von Neuman and data flow, will be presented. The comparison will be made in terms of throughput, flexibility and expandability.

The main drawback of the conventional von Neuman machines is that they are based on an architecture developed with a uniprocessor environment in mind. As a result, the realization of multiprocessor systems based on conventional computers is very difficult and inefficient. That is why data flow machines, with their complete lack of synchronization requirements, are considered so advantageous over con-

ventional von Neuman machines. The proposed architecture makes use of the main characteristic of data flow machines, the presentation of programs as flow graphs and the use of the results of one operation to trigger the execution of another, and, therefore it has all the advantages that data flow machines enjoy over conventional computers.

However, the proposed architecture also has some advantages over conventional data flow machines, at least for signal processing purposes. By taking advantage of the characteristics of signal processing algorithms the necessary data movements have been minimized. Additionally by including the algorithms for the basis functions into the arithmetic processors the control requirements have been greatly reduced making possible the use of centralized control with all its advantages.

On top of these, the proposed architecture is highly modular making any expansions and/or alterations a trivial matter. The addition of any piece of hardware (module or data transfer processor) to an existing system can be done by a simple parallel connection without any changes to the existing hardware and/or software.

5.2 SYSTEM COST VERSUS PERFORMANCE

In this section a cost analysis of the proposed system is presented along with a discussion on the relation between the system cost and performance.

Factors affecting the cost of a multiprocessor system are the number of processors and memory modules, the cost of the interconnection network (switches), and the cost of the control and I/O units. Assuming that \$P is the price of each arithmetic processor, \$M of each memory, \$S of each switch and \$T of each data transfer processor, the cost of each module will be :

$$C_m = P + r(M + S)$$

while the cost of each data transfer processor will be :

$$C_t = T + nrS$$

where

r is the number of memories per module

n is the number of modules in the system

Combining the above we obtain the cost of the system:

$$\begin{aligned} C &= 2(nC_m + kC_t) \\ &= 2(nP + nr(M + S) + kT + knrS) \end{aligned}$$

where

k is the number of data transfer processors

For a microprocessor system it would be a reasonable assumption that $M = P/5$, $S = P/20$ and $T = P$. Therefore the cost equation becomes :

$$C = P(2n + 0.5nr(1 + 0.1k) + k).$$

Figure 5.1 illustrates the cost of the system as a function of the number of arithmetic processors with the number of data transfer processors as a parameter. The number of memories per module was assumed to be 4. Figure 5.2 illustrates the system cost versus performance again with the number of data transfer processors as a parameter. The flow graph of the program used as a load is given in Figure 4.2.

5.3 SYSTEM RESPONSE TIME.

The response time of a signal processing system is very important, especially if the system is to be used in real time applications. In this section the response times of the two jobs mentioned in chapter 4 will be presented and some conclusions will be derived from them. The response times of the jobs shown in Figures 4.1 and 4.2 are given in Figures 5.3 and 5.4 respectively as a function of the number of arithmetic processors with the number of data transfer processors as a parameter.

The dotted line in Figure 5.3 marks the execution time of the job on a uniprocessor. It can be seen that the response time, when the system is fully utilized, is about twice that of the uniprocessor. This was expected since the

number of jobs in the system was set to twice the number of arithmetic processors and therefore each job had to go through a queue of length 2. However, as soon as the utilization of the arithmetic processors drops, there is a steep change to the response time, which soon reaches the response time of the uniprocessor. The utilization of the arithmetic processors, when the response time is within 10% of the uniprocessor value, is found by comparing Figures 5.3 and 4.4 and it turns out to be 70% to 80% depending on the size of the system.

The dotted lines in Figure 5.4 mark the execution time of the job on a uniprocessor and the minimum possible execution time of the job taking full advantage of the inherent parallelism. In this case the response time when the system is fully utilized is only 10% larger than the uniprocessor value. However, as soon as the utilization of the arithmetic processors decreases the response time becomes less than that of the uniprocessor. The final value of the response time depends on the size of the system and is better for larger systems. By comparing Figures 5.4 and 4.8 it is found that the response time reaches its minimum when the utilization of the arithmetic processors has decreased to 80% - 90% depending on the size of the system.

Figure 5.1 System Cost vs System Size
(k is the number of data transfer processors)

Figure 5.2 System Cost vs Throughput
(k is the number of data transfer processors)

Figure 5.3 Response Time of "non-real" time job
vs number of processors

The numbers on the curves represent the number of data transfer processors. The \$ signs represent points where more than one values coincide. The dotted line shows the execution time on a uniprocessor.

Figure 5.4 Response Time of "real" time job
vs number of processors

The numbers on the curves represent the number of data transfer processors. The \$ signs represent points where more than one values coincide. The upper dotted line represents the execution time on a uniprocessor while the lower one represents the minimum possible execution time for the given program

In both cases the response times of larger systems are better than those of smaller systems. For example, in Figure 5.4 the response time of the system, with 130 arithmetic processors and 4 data transfer processors, is almost equal to the minimum possible response time of the job under consideration, while the utilization of the arithmetic processors in this case, as found from Figure 4.8, is about 90%.

The above indicate that the proposed system can be used very efficiently for real time applications. It is important to note here that the simple minded control algorithm used for the simulation is by no means appropriate for a real time environment and a better control algorithm would yield much better results.

5.4 CONCLUSIONS

The purpose of this work was the development of an efficient data flow architecture for digital signal processing. To accomplish that, various available alternatives were examined. The traditional data flow machine was found inefficient due to bottlenecks occurring at several places, and therefore, an alternative was sought. In chapter 1, several alternatives were discussed and their strong and weak points were pointed out. Taking experience from these

architectures and having in mind the specific application of the machine under design, a novel architecture was proposed in chapter 2.

This architecture has all the advantages of conventional data flow machines and in addition it is specifically intended for digital signal processing, and, therefore, it is designed with the peculiarities of digital signal processing algorithms in mind. Except for the non-standard logical description of the machine, a standard ADA description was also included.

In order to verify the feasibility and predicted performance of the proposed architecture, a logical simulation was made which verified that the proposed architecture fulfills all the requirements of a high performance digital signal processor, and, besides, it can be easily expanded without any major changes in the hardware, and with no changes at all in the software. The utilization of the proposed machine proved to be surprisingly high, approaching 100%. Additionally, the proposed architecture is technology transparent, and can be based on any present or future VLSI components, without any need for modification.

Being a data flow machine, the proposed system can extract all inherent parallelism in any given algorithm but,

on the other hand, being a hierarchical system it offers simple control of operation, based on experience gained from conventional computers.

Since, to my knowledge, there are no results available derived from a real, full scale, data flow machine, it is hard to make a direct comparison of the proposed architecture to some generally accepted architecture, in order to derive conclusions. Therefore, it was necessary to use conventional measures, such as number of operations per second, hardware utilization, etc. These measures proved the proposed architecture to be superior to any architecture available today in terms of utilization and throughput.

Further study in this area might include a detailed hardware design and implementation of the proposed system, the analysis and evaluation of the control requirements, the design of a special purpose controller, and the development of an optimum control algorithm.

REFERENCES

[1] S.P.Kartashev, S.I.Kartashev, "Supersystems for the 80's", IEEE Computer, November 1980, pp 11-14.

[2] C.R.Vick, S.P.Kartashev, S.I.Kartashev, "Adaptable Architectures for Supercomputers", IEEE Computer, November 1980, pp 17-35.

[3] C.G.Davis, R.L.Couch, "Ballistic Missile Defense : A Supercomputer Challenge", IEEE Computer, November 1980, pp 37-46.

[4] J.B.Dennis, "Data Flow Supercomputers", IEEE Computer, November 1980, pp 48-56.

[5] E.W.Kozdrowicki, D.J.Theis, "Second Generation of Vector Supercomputers" IEEE Computer, November 1980, pp 71-83.

[6] M.J.Flynn, "Directions and Issues in Architecture and Language", IEEE Computer, October 1980, pp 5-22.

[7] K.P.Gostelow, R.E.Thomas, "Performance of a Simulated Data Flow Computer", IEEE Transactions on Computers, October 1980, pp 905-919.

[8] Tse-Yun Feng, "A Survey of Interconnection Networks", IEEE Computer, December 1981.

[9] G.H.Barnes, S.F.Lundstrom, "Design and Validation of a Connection Network for Many-Processor Multiprocessor Systems" IEEE Computer, December 1981.

[10] A.L.Davis, R.M.Keller, "Data Flow Program Graphs", IEEE Computer, February 1982.

[11] I.Watson, J.Gurd, "A Practical Data Flow Computer", IEEE Computer, February 1982.

[12] H.S.Stone, "Introduction to Computer Architecture", S.R.A. 1980, 1975.

[13] B.A.Bowen, R.J.Buhr, "The Logical Design of Multiple Microprocessor Systems", Prentice Hall Inc. 1980.

[14] P.Grogono, "Programming in PASCAL", A.Wesley 1980.

[15] R.Buhr, D.mackinnon, "Alternatives for the logical design of Signal Processing Systems"

[16] W.R.Brown, "Digital Signal Processing Functions

for VLSI systems design", MEng Thesis, Carleton University,
August 1980.

[17] United States Department of Defense, "Reference
Manual for the ADA Programming Language", July 1980.

[18] A.Tanenbaum, "Computer Networks", Prentice Hall
Inc., 1981.

APPENDIX A

LISTING OF SIMULATION PROGRAM

(*\$L+*)

CONST

```
Maxmemories = 10; { Max number of memories per module }
Maxmodules = 150; { Max number of modules in system }
Maxdmabuses = 4; { Max number of Data Transfer Processors in system }
Maxbacktasknum = 300; { Max number of non-real time tasks }
Setuptime = 0.01; { Time required for recognition of new instruction by Arithmetic processor }
Lratio = 1000.0; { Ratio of Large to normal parameters }
```

TYPE { Type Definitions }

Priority = (High,Low);

Tasktype = (Task1,Task2,Task3,Task4);

Operation = (Hamw, { Hamming Window }
 Fshift, { Frequency Shift }
 Fft, { Fast Fourier Transform }
 Magn, { Magnitude }
 Mf, { Matched Filter }
 Conv, { Convolution }
 Corr, { Correlation }
 Movin, { Move Data In }
 Movout, { Move Data Out }
 Movint); { Move Data From Memory
 to Memory }

Mem = RECORD
 Modn,Memn : Integer
 END;

Template = RECORD
 Opcode : Operation;
 Nextptr : ^Template
 END;

Processor = RECORD
 Free : Boolean;
 Mmr,Mmr2 : Mem; { Memory(ies) Used }
 Tempptr, { Pointers To Templates
 To Be Executed }
 Tempend : ^Template; { Task Executed }
 Ref : ^Tre; { Termination Time of
 present instruction }
 Time : Real
 END;

Tre = RECORD
 Tsk : Tasktype;
 Fptr2,Bptr2 : ^Tre;
 Memo : Mem; { Memory Used }
 Active : Boolean;
 Prio : Priority; { Priority Of The Task }
 Tinit : Real; { Initialization time }
 First : ^Template { ^ To First Template }
 END;

Trep = ^Tre;

Templateptr = ^Template;


```

VAR      {      Variable Definitions      }

Memory : ARRAY[1..Maxmodules,1..Maxmemories] OF Integer;
Proc   : ARRAY[1..Maxmodules] OF Processor;
Dtp    : ARRAY[1..Maxdmabuses] OF Processor;
Nooffree : ARRAY[1..Maxmodules] OF Integer;

Creationtime,      { Time when task will be initialized }
Prtime,            { Present Time }
Nexttime,          { Time after time step }
Pltime,            { Present time high part }
Timestep,          { Time Step size }
Procuti, Lprocuti, { Processors Utilization }
Memuti, Lmemuti,   { Memory Utilization }
Dtputi, Ldtputi,   { Data Transfer Processors Utilization }
Procthr, Lprocthr, { Processor Throughput ( # of MULs ) }
Period,            { Period of real time tasks }
Finaltime,         { Time to stop simulation }
Duration,
Start, Oldtime, Sstime, Throughput, Uti : Real;

Timetable,          { Execution times of basis functions }
Noofmul,             { Number of Multiplications for basis }
                    { functions }
                    : ARRAY[Operation] OF Real;

Found,
Memfull,            { Memory is Full }
Details,            { Do print details of simulation }
Steadyonly : Boolean; { Consider steady state only }
Maxmem,             { Max number of memories used }
Modules,            { Number of processors in system }
Memories,           { Number of memories per module }
Dmabuses,           { Number of Data Transfer Processors }
Backtsknum,         { Number of non-real time tasks }
Tasksin,            { Number of real time tasks }
Branchin,           { Number of secondary tasks }
Tottskout,          { Number of completed real time tasks }
Totbackout,         { Number of completed background tasks }
Totbranchout,       { Number of completed secondary tasks }
Noofsels,           { Number of table lookups that had to }
                    { be done by the controller }

Totm,
Totpr Totdma, Ttot, Oldmaxmem, I,J : Integer;
Hptskb, Lptskb, Ttpr, Tmptskptr : Trep;
Scrptr : ^Template;
Tskrest : ARRAY[Task1..Task4] OF Real;
Brnchcnt : ARRAY[Task3..Task4] OF Integer;
Memr : Mem;
Tskref : ARRAY[Tasktype] OF ^Template;

```

```
PROCEDURE Initialize;
```

```
  VAR
```

```
    I, J : Integer;
```

```
  BEGIN
```

```
    Noofsels := 0;
```

```
    Prtime := 0.0;
```

```
    Creationtime := 0.0;
```

```
    Memfull := FALSE;
```

```
    Maxmem := 0;
```

```
    Oldmaxmem := 0;
```

```
    Ltime := 0;
```

```
    Oldtime := 0;
```

```
    Procuti := 0.0;
```

```
    Lprocuti := 0.0;
```

```
    Memuti := 0.0;
```

```
    Lmemuti := 0.0;
```

```
    Dtputi := 0.0;
```

```
    Ldtputi := 0.0;
```

```
    Procthr := 0.0;
```

```
    Lprocthr := 0.0;
```

```
    Tasksin := 0;
```

```
    Branchin := 0;
```

```
    Tottskout := 0;
```

```
    Totbackout := 0;
```

```
    Totbranchout := 0;
```

```
    Start := 0.0;
```

```
    Tskrest[Task1] := 0.0;
```

```
    Tskrest[Task2] := 0.0;
```

```
    Tskrest[Task3] := 0.0;
```

```
    Tskrest[Task4] := 0.0;
```

```
    Brnchcnt[Task3] := 0;
```

```
    Brnchcnt[Task4] := 0;
```

```
    FOR I := 1 TO Modules DO
```

```
      BEGIN
```

```
        FOR J := 1 TO Memories DO
```

```
          Memory[I, J] := 0;
```

```
          Nooffree[I] := Memories;
```

```
          Proc[I].Free := TRUE
```

```
        END;
```

```
    FOR I := 1 TO Dmabuses DO
```

```
      Dtp[I].Free := TRUE
```

```
  END;
```

```
PROCEDURE Userinterface;
```

```
  VAR
```

```
    I,J,Answer : Integer;
```

```
  BEGIN
```

```
    READLN;
```

```
    REPEAT
```

```
      WRITE(' Number Of Modules = ');
```

```
      READ(Modules);
```

```
    UNTIL (Modules <= Maxmodules) AND (Modules > 0);
```

```
    Memories := Maxmemories;
```

```
    REPEAT
```

```
      WRITE(' Number Of Dma Buses = ');
```

```
      READ(Dmabuses);
```

```
    UNTIL (Dmabuses <= Maxdmabuses) AND (Dmabuses > 0);
```

```
    Backtsknum := 0;
```

```
    Period := 25.0 / Modules;
```

```
    IF Period < 0.85 / Dmabuses
```

```
      THEN
```

```
        Period := 0.85 / Dmabuses;
```

```
    Finaltime := 0.99;
```

```
    Details := FALSE;
```

```
    Steadyonly := TRUE
```

```
  END;
```

```
PROCEDURE Copytask(Tptr : Trep;  
                  Tt : Tasktype);
```

```
  BEGIN
```

```
    Tptr.First := Tskref[Tt]
```

```
  END;
```

```
PROCEDURE Settasks;
```

```
VAR
```

```
    Last : ^Template;
```

```
BEGIN
```

```
    { TASK 1 }
```

```
    NEW(Tskref[Task1]);
```

```
    WITH Tskref[Task1]^ DO
```

```
        BEGIN
```

```
            Opcode := Movin;
```

```
            NEW(Nextptr);
```

```
            Last := Nextptr
```

```
        END;
```

```
    Last^.Opcode := Hamw;
```

```
    NEW(Last^.Nextptr);
```

```
    Last := Last^.Nextptr;
```

```
    Last^.Opcode := Fshift;
```

```
    NEW(Last^.Nextptr);
```

```
    Last := Last^.Nextptr;
```

```
    Last^.Opcode := Fft;
```

```
    NEW(Last^.Nextptr);
```

```
    Last := Last^.Nextptr;
```

```
    Last^.Opcode := Movout;
```

```
    Last^.Nextptr := NIL;
```

```
    { TASK 2 }
```

```
    NEW(Tskref[Task2]);
```

```
    WITH Tskref[Task2]^ DO
```

```
        BEGIN
```

```
            Opcode := Movin;
```

```
            NEW(Nextptr);
```

```
            Last := Nextptr
```

```
        END;
```

```
    Last^.Opcode := Hamw;
```

```
    NEW(Last^.Nextptr);
```

```
    Last := Last^.Nextptr;
```

```
    Last^.Opcode := Fshift;
```

```
    NEW(Last^.Nextptr);
```

```
    Last := Last^.Nextptr;
```

```
    Last^.Opcode := Movint;
```

```
    Last^.Nextptr := NIL;
```

{ TASK 3 }

```
NEW(Tskref[Task3]);
WITH Tskref[Task3]^ DO
  BEGIN
    Opcode := Mf;
    NEW(Nextptr);
    Last := Nextptr
  END;
Last^.Opcode := Movout;
Last^.Nextptr := NIL;
```

{ TASK 4 }

```
NEW(Tskref[Task4]);
WITH Tskref[Task4]^ DO
  BEGIN
    Opcode := Fft;
    NEW(Nextptr);
    Last := Nextptr
  END;
Last^.Opcode := Magn;
NEW(Last^.Nextptr);
Last := Last^.Nextptr;
Last^.Opcode := Movout;
Last^.Nextptr := NIL
END;
```

```
PROCEDURE Setvalues;
```

```
  BEGIN
```

```
    Timetable[Hamw] := 0.15;
    Timetable[Fshift] := 8.0;
    Timetable[Fft] := 6.0;
    Timetable[Magn] := 0.5;
    Timetable[Sti] := 0.2;
    Timetable[Mf] := 10.0;
    Timetable[Conv] := 20.0;
    Timetable[Movin] := 0.2;
    Timetable[Movout] := 0.2;
    Timetable[Movint] := 0.2;
    Noofmuls[Hamw] := 1.0 * 1024.0;
    Noofmuls[Fshift] := 25.0 * 1024.0;
    Noofmuls[Fft] := 20.0 * 1024.0;
    Noofmuls[Magn] := 2.0 * 1024.0;
    Noofmuls[Sti] := 1.0 * 1024.0;
    Noofmuls[Mf] := 28.0 * 1024.0;
    Noofmuls[Conv] := 65.0 * 1024.0
```

```
  END;
```

```
PROCEDURE Createtre(Tskt : Tasktype;
  VAR Elptr : Trep);
```

```
  BEGIN
```

```
    NEW(Elptr);
```

```
    WITH Elptr DO
```

```
      BEGIN
```

```
        Tsk := Tskt;
```

```
        Active := FALSE;
```

```
        Tinit := Ltime + Prtime / 1000.0;
```

```
        IF Tsk = Task1
```

```
          THEN
```

```
            Prio := Low
```

```
          ELSE
```

```
            Prio := High
```

```
        END;
```

```
        Copytask(Elptr, Tskt)
```

```
      END;
```

```
PROCEDURE Dealwithtre(Treptr : Trep);
```

```
  BEGIN
```

```
    IF Treptr^.Prio = Low
```

```
    THEN
```

```
      BEGIN
```

```
        Copytask(Treptr, Task1);
```

```
        Tskrest[Task1] := Tskrest[Task1] + Ltime + Prtime  
                          / 1000.0 - Treptr^.Tinit;
```

```
        Totbackout := Totbackout + 1;
```

```
        Treptr^.Tinit := Ltime + Prtime / 1000.0;
```

```
        Treptr^.Active := FALSE
```

```
      END
```

```
    ELSE
```

```
      BEGIN
```

```
        Treptr^.Fptr2^.Bptr2 := Treptr^.Bptr2;
```

```
        Treptr^.Bptr2^.Fptr2 := Treptr^.Fptr2;
```

```
        IF Treptr^.Tsk = Task2
```

```
        THEN
```

```
          BEGIN
```

```
            Tasksin := Tasksin - 1;
```

```
            Tottskout := Tottskout + 1;
```

```
            Tskrest[Task2] := Tskrest[Task2] + Ltime +  
                              Prtime / 1000.0 -  
                              Treptr^.Tinit;
```

```
            Branchin := Branchin + 2
```

```
          END
```

```
        ELSE
```

```
          BEGIN
```

```
            Branchin := Branchin - 1;
```

```
            Brnchcnt[Treptr^.Tsk] :=
```

```
              Brnchcnt[Treptr^.Tsk] + 1;
```

```
          + 1;
```

```
          Tskrest[Treptr^.Tsk] := Tskrest[Treptr^.Tsk]  
                                + Ltime +  
                                Prtime/1000.0 -  
                                Treptr^.Tinit;
```

```
          Totbranchout := Totbranchout + 1
```

```
        END;
```

```
        Dispose(Treptr)
```

```
      END;
```

```
    END;
```

```

PROCEDURE Copytemplates(VAR Source : Templateptr;
                        VAR Destination : Templateptr;
                        VAR Endptr : Templateptr);

```

```

BEGIN
  Destination := Source;
  WHILE (Source # NIL) AND
    (ORD(Source^.Opcode) < ORD(Movin)) DO
    BEGIN
      Endptr := Source;
      Source := Source^.Nextptr
    END
  END;

```

```

{ Find Module With Most Free Memories }

```

```

PROCEDURE Selmem(VAR Memr : Mem);

```

```

VAR
  Max, Temp, I, J : Integer;

BEGIN
  Noofsels := Noofsels + 1;
  Max := 0;
  FOR I := 1 TO Modules DO
    BEGIN
      Temp := Nooffree[I];
      IF Temp > Max
      THEN
        BEGIN
          Max := Temp;
          Memr.Modn := I
        END
      END;
      IF Max > 0
      THEN
        BEGIN
          I := 1;
          WHILE Memory[Memr.Modn, I] # 0 DO
            I := I + 1;
          Memr.Memn := I;
          Nooffree[Memr.Modn] := Nooffree[Memr.Modn] - 1
        END
      ELSE
        Memfull := TRUE
      END;
    END;
  END;

```


PROCEDURE Timeadvance;

BEGIN

Nexttime := Creationtime;

FOR I := 1 TO Modules DO

IF NOT Proc[I].Free AND (Nexttime > Proc[I].Time)
THEN

Nexttime := Proc[I].Time;

FOR I := 1 TO Dmabuses DO

IF (NOT Dtp[I].Free) AND (Nexttime > Dtp[I].Time)
THEN

Nexttime := Dtp[I].Time;

Timestep := Nexttime - Prtime;

Prtime := Nexttime

END;

```

PROCEDURE Assigntransfer(VAR Ptskb :trep;
                        VAR Pr : Processor);
VAR
  Ttptr : ^Tre;
  Memr : Mem;

BEGIN
  WITH Pr DO BEGIN
    Ttptr := Ptskb^.Fptr2;
    WHILE (Ttptr # Ptskb) AND NOT Found
      AND NOT Memfull DO BEGIN
        WITH Ttptr^ DO
          IF (ORD(First^.Opcode) >= ORD(Movin))
            AND (NOT Active)
          THEN
            BEGIN
              IF First^.Opcode = Movout
                THEN
                  Memr := Memo
                ELSE
                  Selmem(Memr);
              IF NOT Memfull
                THEN
                  WITH Dtp[1] DO BEGIN
                    Tempptr := First;
                    Time := Prtime + Setuptime +
                      Timetable[Tempptr^.Opcode];
                    Dtputi := Dtputi - Setuptime;
                    Free := FALSE;
                    IF First^.Opcode = Movint
                      THEN
                        Mmr2 := Memo;
                    Mmr := Memr;
                    Ref := Ttptr;
                    IF First^.Opcode = Movout
                      THEN
                        Memory[Memr.Modn,Memr.Memn] := 5
                      ELSE
                        Memory[Memr.Modn,Memr.Memn] := 1;
                    First := First^.Nextptr;
                    Tempend := Tempptr;
                    Active := TRUE;
                    Found := TRUE
                  END
                END;
            Ttptr := Ttptr^.Fptr2
          END;
        Memfull := FALSE
      END
    END;
  END;
END;

```

```
PROCEDURE Assignwork(Ptskb : Trep);
```

```
VAR
```

```
  Ttptr : Trep;
```

```
BEGIN
```

```
  Ttptr := Ptskb^.Fptr2;
```

```
  WHILE Ttptr # Ptskb DO BEGIN
```

```
    WITH Ttptr DO
```

```
      IF (NOT Active) And
```

```
        (ORD(First^.Opcode) < ORD(Movin))
```

```
      THEN IF Proc[Memo.Modn].Free
```

```
        THEN
```

```
          WITH Proc[Memo.Modn] DO BEGIN
```

```
            Free := FALSE;
```

```
            Mmr := Memo;
```

```
            Memory[Mmr.Modn, Mmr.Memn] := 3;
```

```
            Ref := Ttptr;
```

```
            Copytemplates(First, Tempptr, Tempend);
```

```
            Time := Pptime + Setuptime +
```

```
              Timetable[Tempptr^.Opcode];
```

```
            Procuti := Procuti - Setuptime;
```

```
            Active := TRUE
```

```
          END;
```

```
  Ttptr := Ttptr^.Fptr2
```

```
END
```

```
END;
```

```
PROCEDURE Timeadjust;
```

```
BEGIN
```

```
  IF Pptime > 125.0
```

```
  THEN
```

```
    BEGIN
```

```
      Pptime := Pptime - 125.0;
```

```
      Creationtime := Creationtime - 125.0;
```

```
      FOR I := 1 TO Modules DO
```

```
        IF NOT Proc[I].Free
```

```
        THEN
```

```
          Proc[I].Time := Proc[I].Time - 125.0;
```

```
      FOR I := 1 TO Dmabuses DO
```

```
        IF NOT Dtp[I].Free
```

```
        THEN
```

```
          Dtp[I].Time := Dtp[I].Time - 125.0;
```

```
      Ltime := Ltime + 0.125
```

```
    END
```

```
END;
```

```
PROCEDURE Createtsk(T : Tasktype;
                    VAR B : Trep);
```

```
  BEGIN
    Createtre(T, Tmptrskptr);
    Tmptrskptr^.Fptr2 := B;
    Tmptrskptr^.Bptr2 := B^.Bptr2;
    B^.Bptr2^.Fptr2 := Tmptrskptr;
    B^.Bptr2 := Tmptrskptr;
  END;
```

6

```
PROCEDURE Adjustvar(VAR Varname : Real;
                    VAR Lvarname : Real);
```

```
  BEGIN
    IF Varname > 125.0
    THEN
      BEGIN
        Varname := Varname - 125.0;
        Lvarname := Lvarname + 0.125
      END
    END;
  END;
```

PROCEDURE Statistics;

VAR

I, J : Integer;

BEGIN

Totm := 0;

Totpr := 0;

Totdma := 0;

FOR I := 1 TO Modules DO BEGIN

Ttot := 0;

FOR J := 1 TO Memories DO

IF Memory[I, J] # 0

THEN

BEGIN

Ttot := Ttot + 1;

Totm := Totm + 1

END;

IF Maxmem < Ttot

THEN

Maxmem := Ttot;

IF NOT Proc[I].Free

THEN

Totpr := Totpr + 1

END;

FOR I := 1 TO Dmabuses DO

IF NOT Dtp[I].Free

THEN

Totdma := Totdma + 1;

Procuti := Procuti + Totpr * Timestep;

Adjustvar(Procuti, Lprocuti);

Memuti := Memuti + Totm * Timestep;

Adjustvar(Memuti, Lmemuti);

Dtputi := Dtputi + Totdma * Timestep;

Adjustvar(Dtputi, Ldtputi);

IF (Maxmem > Oldmaxmem) AND Steadyonly AND (Maxmem <= 2)

THEN

BEGIN

Procthr := 0.0;

Lprocthr := 0.0;

Procuti := 0.0;

Lprocuti := 0.0;

Memuti := 0.0;

Lmemuti := 0.0;

Dtputi := 0.0;

Ldtputi := 0.0;

Oldmaxmem := Maxmem;

Start := Ltime + (Ptime - Timestep) / 1000.0

END

END;

```

BEGIN      { Main Program }
    Setvalues;

    Settasks;

    Userinterface;

    Initialize;

    { Create Two Empty Task Rings }

    NEW(Hptskb);
    Hptskb^.Fptr2 := Hptskb;
    Hptskb^.Bptr2 := Hptskb;
    NEW(Lptskb);
    Lptskb^.Fptr2 := Lptskb;
    Lptskb^.Bptr2 := Lptskb;

    { Create Backtsknum Background Tasks }

    IF Backtsknum > 0
    THEN
        FOR I := 1 TO Backtsknum DO
            Createtsk(Task1,Lptskb);

    { Main Program Loop }

REPEAT

    { Add New Real Time Tasks }

    IF Pptime >= Creationtime
    THEN
        BEGIN
            Createtsk(Task2,Hptskb);
            Tasksin := Tasksin + 1;
            Creationtime := Creationtime + Period
        END;

```

```
{ Remove Executed Templates }
```

```
{ A) From Data Transfer Processors }
```

```
FOR I := 1 TO Dmabuses DO
```

```
  WITH Dtp[I] DO
```

```
    IF NOT Free AND (Time <= Prtime)
    THEN
```

```
      BEGIN
```

```
        Free := TRUE;
```

```
        IF Tempptr^.Opcode = Movout
        THEN
```

```
          BEGIN
```

```
            Memory[Mmr.Modn,Mmr.Memn] := 0;
```

```
            Nooffree[Mmr.Modn] := Nooffree[Mmr.Modn] ~
            + 1
```

```
          END
```

```
        ELSE
```

```
          Memory[Mmr.Modn,Mmr.Memn] := 2;
```

```
{ IF Fork Point THEN }
```

```
  IF Tempptr^.Opcode = Movint
  THEN
```

```
    BEGIN
```

```
      Memory[Mmr2.Modn,Mmr2.Memn] := 2;
```

```
{ Create Branch Number 1 }
```

```
      Createtsk(Task3,Hptskb);
```

```
      Tmptskptr^.Memo := Mmr;
```

```
{ Create Branch Number 2 }
```

```
      Createtsk(Task4,Hptskb);
```

```
      Tmptskptr^.Memo := Mmr2
```

```
    END;
```

```
    Ref^.Active := FALSE;
```

```
    Ref^.Memo := Mmr;
```

```
    IF Ref^.First = NIL
```

```
    THEN
```

```
      Dealwithtre(Ref);
```

```
    Tempptr := NIL
```

```
  END;
```

```

{ B) From Processing Elements }

FOR I := 1 TO Modules DO
  WITH Proc[I] DO
    IF NOT Free AND (Time <= Prtime)
    THEN
      BEGIN
        Procthr := Procthr +
          Noofmul*[Tempptr^.Opcode];
        IF Procthr > 1000000.0
        THEN
          BEGIN
            Procthr := Procthr - 1000000.0;
            Lprocthr := Lprocthr + 1
          END;
        IF Tempptr # Tempend
        THEN
          BEGIN
            Tempptr := Tempptr^.Nextptr;
            Time := Prtime +
              Timetable[Tempptr^.Opcode]
          END
        ELSE
          BEGIN
            Free := TRUE;
            Memory[Mmr.Modn,Mmr.Memn] := 4;
            Ref^.Memo := Mmr;
            Ref^.Active := FALSE;
            IF Ref^.First = NIL
            THEN
              Dealwithtre(Ref);
            Tempptr := NIL
          END
        END;
      END;
    END;
  END;

```



```
{ Assign Transfers To Data Transfer Processors }
```

```
FOR I := 1 TO Dmabuses DO
```

```
  IF Dtp[I].Free
```

```
    THEN
```

```
      BEGIN
```

```
        { A) From High Priority Tasks }
```

```
          Found := FALSE;
```

```
          Assigntransfer(Hptskb,Dtp[I]);
```

```
        { B) From Low Priority Tasks }
```

```
          IF NOT Found
```

```
            THEN
```

```
              Assigntransfer(Lptskb,Dtp[I])
```

```
            END;
```

```
{ Assign Work To Processing Elements }
```

```
{ A) From High Priority Tasks }
```

```
Assignwork(Hptskb);
```

```
{ B) From Low Priority Tasks }
```

```
Assignwork(Lptskb);
```

```
{ Update Time Counter }
```

```
Timeadjust;
```

```
Timeadvance;
```

```
{ Compute Statistics }
```

```
Statistics;
```

```

IF Details AND ((Ltime # Oldtime) OR (Ltime = 0))
THEN
  BEGIN
    WRITE(Prtime-timestep);
    WRITE(Procuti,Memuti,Dtputi,Tasksin:5);
    WRITELN(Branchin:5,Maxmem:5);
    WRITE(Ltime);
    WRITE(Lprocuti,Lmemuti,Ldtputi,Tottskout:5);
    WRITELN(Totbranchout:5,Totbackout:5);
    FOR I := 1 TO Modules DO
      BEGIN
        WRITE(Proc[I].Free);
        FOR J := 1 TO Memories DO
          WRITE(Memory[I,J]:4);
        WRITELN
      END;
    FOR I := 1 TO Dmabuses DO
      WRITE(Dtp[I].Free);
    WRITELN;
    Oldtime := Ltime;
  END;
UNTIL (Ltime > Finaltime);

WRITELN;
WRITELN('      System Parameters');
WRITELN(Modules:5,' Modules');
WRITELN(Memories:5,' Memories Per Module');
WRITELN(Dmabuses:5,' Data Transfer Processors');
WRITELN;
WRITELN('      Load Parameters');
WRITE(Backtsknum:5,);
WRITELN(' Background Non-real Time Tasks');
WRITE(1000.0 / Period:7:0);
WRITELN(' Real Time Tasks Initialized Every Second');
WRITELN;
Duration := Ltime + Prtime / 1000.0;
WRITELN(' Duration Of Run = ',Duration:7:3,' Seconds');
Sstime := Duration - Start;
WRITE(' System In Steady State For ',Sstime:7:3);
WRITELN(' Seconds');
WRITE(' Steady State Reached In',Start:7:3);
WRITELN(' Seconds');
Uti := (Lprocuti + Procuti / 1000.0) / Modules
      / Sstime * 100.0;
Throughput := (Lprocthr + Procthr / 1000000.0) / Sstime;

```

```

WRITELN(' Processor Utilization = ',Uti:6:2,' %');
Uti := (Lmemuti + Memuti / 1000.0) / Modules / Maxmem
      / Sstime * 100.0;
WRITELN(' Memory Utilization = ',Uti:6:2,' %');
Uti := (Ldtputi + Dtputi / 1000.0) / Dmabuses
      / Sstime * 100.0;
WRITE(' Data Transfer Processor Utilization = ');
WRITELN(Uti:6:2,' %');
WRITELN;
WRITE(' Up To',Maxmem:4);
WRITELN(' Memories Have Been Used At Any Time');
WRITE(Tottskout:5,);
WRITELN(' Real Time Tasks Have Been Executed');
WRITE(Totbackout:5);
WRITELN(' Background Tasks Have Been Executed');
WRITE(Totbranchout:5);
WRITELN(' Branches Have Been Completed');
WRITE(Tasksin:5);
WRITELN(' Real Time Tasks Remain In The System');
WRITE(Backtsknum:5);
WRITELN(' Background Tasks Remain In System');
WRITELN(Branchin:5,' Branches Remain In The System');
WRITELN;
WRITE(' Throughput Achieved = ',Throughput:10:3);
WRITELN(' Million Multiplications Per Second');
WRITELN;
IF (Totbackout > 0)
  THEN
    BEGIN
      WRITE(' Average Response Time Of Task1 = ');
      WRITE(Tskrest[Task1] / Totbackout * 1000.0 :8:2);
      WRITELN(' Msec')
    END;
  WRITE(' Average Response Time Of Task2 = ');
  WRITE(Tskrest[Task2] / Tottskout * 1000.0 :8:2);
  WRITELN(' Msec');
  WRITE(' Average Response Time Of Task3 = ');
  WRITE(Tskrest[Task3] / Brnchcnt[Task3] * 1000.0 :8:2);
  WRITELN(' Msec');
  WRITE(' Average Response Time Of Task4 = ');
  WRITE(Tskrest[Task4] / Brnchcnt[Task4] * 1000.0 :8:2);
  WRITELN(' Msec');
  WRITELN;
  WRITELN(Noofsels:5,' Table Lookups Were Needed');
  FOR I := 1 TO 20 DO
    WRITELN;
  WRITELN(' JOB TERMINATED');
  WRITELN;
  WRITELN;
END.

```

END

0.610.6.8.3

FIN